Optimisation of Partitioned Temporal Joins

Thomas F. Zurek

Doctor of Philosophy
University of Edinburgh
1997

To Isabel, my wife.

Abstract

Joins are the most expensive and performance-critical operations in relational database systems. In this thesis, we investigate processing techniques for joins that are based on a temporal intersection condition. Intuitively, such joins are used whenever one wants to match data from two or more relations that is valid at the same time.

This work is divided into two parts. First, we analyse techniques that have been proposed for equi-joins. Some of them have already been adapted for temporal join processing by other authors. However, hash-based and parallel techniques – which are usually the most efficient ones in the context of equi-joins – have only found little attraction and leave several temporal-specific issues unresolved. Hash-based and parallel techniques are based on explicit symmetric partitioning. In the case of an equi-join condition, partitioning can guarantee that the relations are split into *disjoint* fragments; in the case of a temporal intersection condition, partitioning usually results in *non-disjoint* fragments with a large number of tuples being replicated between fragments. This causes a considerable overhead for partitioned temporal join processing. This problem is an instance of the 'min-max dilemma': minimising the number of replicated tuples means minimising the number of fragments, thus minimising the degree of parallelism – however, increasing the number of fragments and therefore the degree of parallelism also increases the number of tuple replications. We analyse this problem and show that there is an algorithm of polynomial time complexity that computes an optimal solution for the interval partitioning problem (IP). This result concludes the analytical part.

In the second, the synthetical part of this work, we focus on the conclusions that can be drawn from the results of the first part. We propose and develop an optimisation process that

- analyses the temporal relations that participate in a temporal join,
- proposes several possible partitions for these relations,
- analyses these partitions and predicts their performance implications on the basis of a parameterised cost model, and

- chooses the cheapest partition to process the temporal join.

We also show how this process can be efficiently implemented by using a new index structure, called the IP-table.

The thesis is concluded by a thorough experimental evaluation of the optimisation process and a chapter that shows the suitability of IP-tables in a wider context of temporal query optimisation, namely using them to estimate selectivities of temporal join conditions.

Acknowledgements

After over 1000 days of PhD research, around 500 pages of thesis, paper and report writing, I can finally add the final and the most enjoyable part: it is these lines in which I can thank the many people that have enabled me to produce this work by providing the fruitful environment that I have had in the last three years.

First of all, I have to thank Isabel, my wife, for her endless patience and support and for cheering me up when things did not run as smoothly as my *cabeza cuadrada* would have wished. Furthermore I would like thank my parents and my sister, not only for their financial support during these years but also for confiding in me and my abilities, especially at times when I was not so sure about them. Then, there are the many friends that I have met during that time and who have made this period so pleasant, fruitful and so rich in experience. To mention a few: Cristina Boeres (my second sister) and Vinod Rebello, Ana Goldenberg and Luis Araujo ("love is beautiful"), Beate and Thomas Kleymann ("squash is beautiful"), Olga Savasta and Victor Varela ("los padrinos"), Nils Knafla and Frank Yang (my officemates), Martin Reddy, Rob Payne, Marcus Marr, Mike Galloway, Alistair Ewing and so many more who would deserve to be mentioned. You all contributed in one or the other way, through technical discussions, sharing experience or 'simply' by your company and your friendship. To that list I would like to add Ulf Schwitzke, my best man in many ways.

My special thanks go to Peter Thanisch who not only contributed so many things in the numerous discussions that we had about this work but who also convinced me to start with the PhD in the first place. Supervising a PhD student is not always an easy and pleasant job and there are thousands of other things that one has to pursue. Although things not always went as Peter or I would have wished, I always knew that he tried his best. I also enjoyed playing in the many football matches.

Thanks to all of you!

Declaration

I declare that this doctoral thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text. The following articles were published during my period of research. Certain material and concepts from these publications will necessarily be presented within the body of this work.

1. Zurek, T. (1996). *Parallel Temporal Nested-Loop Joins.* Technical Report ECS-CSG-20-96, Dept. of Computer Science, Edinburgh University.

2. Norman, M., Zurek, T., and Thanisch, P. (1996). *Much Ado about Shared-Nothing.* SIGMOD Record, Vol. 25, No. 3, pages 16–21.

3. Zurek, T. (1997). *Parallel Temporal Joins.* In "Datenbanksysteme in Büro, Technik und Wissenschaft" (BTW), German Database Conference, Ulm, Germany, pages 269–278. Springer Verlag.

4. Zurek, T. (1997). *Optimal Interval Partitioning for Temporal Databases.* In Proc. of the 3rd BIWIT Workshop, Biarritz, France, pages 140–147. IEEE Computer Society Press.

5. Zurek, T. (1997). *Optimisation of Partitioned Temporal Joins.* In Proc. of the 15th BNCOD Conference, London, UK, LNCS 1271, pages 101–115. Springer Verlag.

6. Zurek, T. (1997). *Parallel Processing of Temporal Joins.* To appear in 'Informatica', ISSN 0350-5596.

(Thomas Zurek)

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Recent years have seen an increasing number of technological and economical developments that affect the way in which database systems are used. Data warehousing, data mining, geographical and other information systems, e.g. on the internet, have added new requirements with respect to data modeling and processing performance. As a consequence, many new complex data types, such as spatial, temporal, audio and video data, have been introduced into database management system software. This process has been supported by technological progress, such as the advent of affordable, off-the-shelf parallel hardware which has been widely driven by the high demands of commercial database applications.

In this development, the relational data model plays a key role: on the one hand, it can be relatively easily enhanced to provide the new data types and, on the other hand, it provides a lot of opportunities to exploit new technology, in particular parallelism. The key to the success of parallelism in database technology is that it is largely invisible to the end user and database applications programmer. The team in the database management system vendor's research and development lab provides the basic, general-purpose techniques and the local database administrator fine-tunes the systems, based on characteristics of the local installation's data. The key issue in the way temporal data is introduced is that this present arrangement of hiding parallelism is continued.

In this thesis, we want to contribute to building the connection between new data types and new technology. We will look at the possibility of how time interval data can be joined in parallel by partitioning the time interval data over several processing nodes. Data partitioning is a key issue in parallel database systems. I/O parallelism, for example, is based on physically parti-

tioning data over a large number of disks. This overcomes the ever increasing gap between CPU speed and I/O bandwidth[1]. On the processing side, data partitioning provides the opportunity to magnify the raw computing power of individual commodity processors by partitioning a workload into several portions which can be processed concurrently.

However, data partitioning is not only beneficial for parallel but for sequential join processing. In many situations, it also reduces the amount of processing that is necessary to complete the task. In the case of join processing, for example, we can separate tuples that cannot possibly join by partitioning the data over the join attribute values[2]. Therefore, data partitioning provides advantages in a parallel *and* a sequential environment. We pay attention to this fact by looking at *partitioned temporal join processing*, thus considering data partitioning in a wider context although parallelism will be the main focus.

Furthermore, we stress that our work is not restricted to time intervals but applies to interval data in general. However, interval data is mainly used in the context of temporal database applications and we can derive many requirements and constraints by looking at temporal scenarios. Therefore, we will focus on time intervals.

Before digging deeper into partitioned temporal join processing we want to illustrate some of the temporal- and interval-specific problems by an example.

An Example

For this purpose, the two simple temporal relations of figure 1.1 are used: one holds cities and periods during which the play 'Hamlet' is performed in the respective city, the other does the same for the play 'Faust'. For simplicity, we use integers to denote dates.
If we want to find the cities in which both plays are performed then we can do this by computing the join

$$\textit{Hamlet} \bowtie_C \textit{Faust}$$

with the join condition

$$C \equiv \textit{Hamlet.City} = \textit{Faust.City}$$

This is called an *equi-join* because C is based on the equality predicate. To compute this join in parallel we can partition the two tables by using the values

[1]While CPU clock speeds double every 18 month on average (i.e. a 60% increase per year), the bandwidth of single-disk I/O devices increases by around 10% every year [Gray, 1995].

[2]For a detailed discussion, we refer the reader to chapter 3 where several partitioning methods, such as sorting and hashing, are described and analysed.

Relation **Hamlet**		
City	Start	End
Berlin	3	8
Dublin	2	10
London	4	9
Madrid	1	8
Oslo	6	10
Vienna	1	10

Relation **Faust**		
City	Start	End
Bern	1	4
London	1	4
Munich	7	9
Oslo	2	5
Paris	1	3
Rome	7	10

Figure 1.1: An example of two temporal relations.

of the city attributes. Figure 1.2 shows an example for partitioning the tables into three fragments respectively and thus into three smaller and independent joins. We note that the partitioning process produced *disjoint* fragments respectively. Each of the three joins can be processed on different nodes – referred to as (N1), (N2) and (N3) – in parallel.

If we want to find those periods during which both plays are performed, irrespective of the location of the performances, then we need a *temporal intersection join* between the two relations. The join condition is

$$C \equiv \textit{TIMESTAMP(Hamlet) intersects TIMESTAMP(Faust)}$$

in this case. Similarly to the equi-join above, the temporal join can be processed in parallel. This time, however, the tables have to be partitioned over the interval timestamps. Figure 1.3 shows an example of partitioning the tables into three fragments respectively. We note that the fragments are *not disjoint* in this case and tuples are replicated. This causes an overhead not only because of the effort spent on the replication itself but also because of the additional work imposed on the joining of the fragments.

The problem of intervals overlapping partition breakpoints not only makes it difficult to choose appropriate breakpoints but also makes it delicate to determine the number of breakpoints: in order to reduce the number of overlaps one has to reduce the number of breakpoints but in order to increase the number of partial joins – e.g. in order to increase the degree of parallelism – one has to increase the number of breakpoints. Thus there are two contrary effects associated with interval partitioning; note that the first one does not exist in the case of an equi-join. This indicates that it is not straightforward to find the optimal trade-off partition between minimal overlaps and maximal parallelism. Apart from that, one has to expect further cost constraint given by the actual

hardware platform and the join algorithm.

1.2 Research Goal

In this thesis, we investigate and elaborate mechanisms to optimise partitioned temporal join processing. To that end, we have to consider the problem of tuple replication and investigate how near-optimal partitions can be derived. As optimal partitions might only be found by an exhaustive or at least a very expensive search, we therefore have to investigate whether this is really the case, and thus to analyse the complexity of the problem. Once this has been established, we have to look for heuristics for efficiently finding near-optimal partitions for temporal join processing.

In this thesis, we focus on the intersection of intervals as the principal temporal join condition (see previous example). Many other temporal join conditions, such as an interval being contained in an other interval, are specialisations of this intersection condition. These specialisations allow several performance enhancing optimisation, e.g. the restriction of tuple replication to a certain subset of the participating relations [Leung and Muntz, 1992]. However, many of them still suffer from the same problems as the more general intersection join. In most cases, such as the various types of overlap joins, the contain join and the during join, for example, tuple replication cannot totally be abandoned despite being restricted to only one of the participating relations. Partitioned processing of theses joins can therefore still benefit from the optimisations that we propose. In order to provide a clear distinction between the optimisations that can be based on the more specialised join condition and the optimisations that are applicable to all intersection joins we focus on the intersection join condition and regard our work as complementary to that by Leung and Muntz.

In this thesis, we consider join predicates that involve the intervals associated with the tuples. There might be other, non-interval attributes involved. However, we want to investigate the possibilities arising from partitioning over the interval attributes and therefore concentrate on them, thereby ignoring the parts of the join predicate that are not relevant for partitioning. This also helps to distinguish between optimisation and partitioning methods that are available for predicates over atomic attributes – such methods have already been the focus of a large numbers of papers – and those involving interval attributes – these are investigated in the context of this thesis as they have not

found any attention yet. In the future, one will need to investigate the tradeoff between these two sets of techniques in order to provide a query optimiser with a guideline of how to choose the most appropriate and efficient technique. However, developing such a guideline here in the context of interval partitioning, however, would go beyond the scope of a single thesis.

Hamlet$_1$ $\bowtie_C$ **Faust**$_1$

Hamlet			Faust		
Berlin	3	8	Bern	1	4
Dublin	2	10	London	1	4
London	4	9			

(N1) City starting with A – L

Hamlet$_2$ $\bowtie_C$ **Faust**$_2$

Hamlet			Faust		
Madrid	1	8	Munich	7	9
Oslo	6	10	Oslo	2	5

(N2) City starting with M – O

Hamlet$_3$ $\bowtie_C$ **Faust**$_3$

Hamlet			Faust		
Vienna	1	10	Paris	1	3
			Rome	7	10

(N3) City starting with P – Z

Figure 1.2: Example of processing an equi-join in parallel.

Hamlet$_1$ $\bowtie_C$ **Faust**$_1$

Hamlet			Faust		
Berlin	3	8	Bern	1	4
Dublin	2	10	London	1	4
Madrid	1	8	Oslo	2	5
Vienna	1	10	Paris	1	3

(N1) Timestamp interval intersecting with [1,3]

Hamlet$_2$ $\bowtie_C$ **Faust**$_2$

Hamlet			Faust		
Berlin	3	8	Bern	1	4
Dublin	2	10	London	1	4
London	4	9	Oslo	2	5
Madrid	1	8			
Vienna	1	10			

(N2) Timestamp interval intersecting with [4,5]

Hamlet$_3$ $\bowtie_C$ **Faust**$_3$

Hamlet			Faust		
Berlin	3	8	Munich	7	9
Dublin	2	10	Rome	7	10
London	4	9			
Madrid	1	8			
Oslo	6	10			
Vienna	1	10			

(N3) Timestamp interval intersecting with [6,10]

Figure 1.3: Example of processing a temporal join in parallel

1.3 Synopsis

This thesis comprises eleven chapters apart from this introduction. They can be divided into two major parts:

- the *analytical part* in chapters 2 to 6, in which we motivate the research problem, review the literature and establish the relevance of this research. The results of this analysis lead to the creation of an optimisation process which is the starting point of
- the *synthetical part* in chapters 7 to 11, in which the optimisation process is elaborated and evaluated.

We now give an overview over these two parts.

Analytical Part

In chapter 2, we give an introduction to the area of temporal databases. To this end, we explain basic concepts and notations that will be used throughout this thesis. Temporal databases are contrasted with conventional databases in order to elaborate the temporal-specific research issues. One of these is lack of efficiency of performance-critical temporal operators, such as the temporal join. The latter will be the focus of this work. Finally, we look at the relevance of temporal databases in the commercial world.

In chapter 3, the huge variety of general join techniques is described. The join operation has been intensively analysed by a large number of researchers. However, most of the algorithms that have been proposed are tuned to perform well with equi-joins, as empirical research suggests that equality conditions appear in over 90% of joins in conventional database processing. Usually, temporal join conditions are based on nonequi-join conditions. Therefore, one needs to investigate how well equi-join techniques can cope with nonequi-joins. This problem is the focus of chapter 4. Here, we review the literature and describe the many adaptions that have been proposed for temporal join processing. It emerges that parallel and hash join techniques – usually the most efficient techniques – behave in a significantly different way when applied to temporal joins. The basic problem is that most temporal join conditions require tuples to be replicated between relation fragments. This is not the case for equi-join conditions. Tuple replications, however, cause an overhead in join processing. It is a major task of this work to investigate and to tackle this overhead.

To this end, we analyse the problem of partitioning a collection of intervals in such a way that the resulting fragments have a limited size while the number of necessary tuple replications is minimised. This is called the interval partitioning (IP) problem. The analysis provides a variety of interesting results that are used at various stages of this thesis.

The analytical part is concluded in chapter 6 which summarises the main results. These lead to the creation of a process that optimises partitioned temporal join processing by choosing the most appropriate partition for the data. The structure of the remainder of the thesis is mainly based on the structure of this process.

Synthetical Part

In chapter 7, we look at the first stage of the optimisation process, namely the stage of data analysis. Along with the information that describe the characteristics of the timestamp intervals, we propose to maintain a new metadata-structure called the *IP-table*. We give a formal definition of an IP-table, compare it to alternative approaches, such as using data samples, show how the size of an IP-table can be decreased by *condensing* timepoints, provide algorithms for the maintenance of IP-tables and finally describe how two or more IP-tables can be merged. All these operations are required in the context of the optimisation process.

In chapter 8, we develop a detailed performance model for partitioned temporal join processing. This is done in three stages: firstly, we discuss issues concerning the underlying hardware architecture and create an *architectural model*; secondly, we describe how a partitioned temporal join is processed on the architectural model – this is referred to as the *temporal join processing model*; finally, we derive a detailed *cost model* for temporal join processing. The chapter is concluded by a simple experimental evaluation that provides some insight into the characteristics of the performance model. This information helps us to identify certain performance-critical issues.

Chapter 9 discusses several families of partitioning strategies. Every strategy can provides a partition candidate for the optimisation process. Obviously, there is a huge variety of such strategies as each one has a certain optimisation goal as its target.

In chapter 10, we conduct a thorough experimental evaluation of the techniques and methods that have been developed in preceding chapters. This gives some insight into the characteristics of the partitioning strategies and

their performance impact.

In chapter 11, we show that IP-tables can be used not only for partitioning purposes, but also to estimate the selectivity of temporal joins. In conventional database management systems, selectivity estimation is an important tool for a query optimiser to distribute the load and balance query processing.

Finally, the thesis is concluded in chapter 12 which summarises the work and its major contributions and holds an outlook to future work.

Chapter 2

Temporal Databases

Time is an important entity in everyday life. Everyone of us has to check calendars, diaries, timetables etc. on a daily routine. Therefore it is not surprising that temporal information has also made its way into many information management systems. A recent reminder of this fact are the many discussions around the implication of what is called the "date crisis", the "year-2000-bug" or the "millenium timebomb", as in [Glass, 1997], [Clark, 1997], [Uhlig, 1997] and many more.

In this chapter, we focus on time data stored in databases. It is intended to serve as a general introduction to the research area of temporal databases. Furthermore, it will introduce some basic concepts that are used throughout the remainder of this thesis.

2.1 Introduction

Temporal Databases

Temporal databases store temporal data, i.e. data that is time-dependent (time-varying). Typical temporal database scenarios and applications are the following:

- Economical data is frequently time-dependent: share prices, exchange rates, interest rates, company profits etc. vary over time. This means that we need to store not only the respective value but also an associated date or a time period for which the value is valid. Typical queries, for example, are
  - Give me last month's history of the Dollar - Pound Sterling exchange rate.
  - Give me the share prices of the NYSE on October 17, 1996.

More sophisticated analysis might want to correlate interest rates and exchange rate or share prices trends. This means that an interest rate value has to be related to an exchange rate value using the date or period for which the values are valid: they have to be valid during the same period of time in this example.

- Many companies offer products whose prices vary over time. Daytime telephone calls, for example, are usually more expensive than evening or weekend calls. Travel agents, airlines or ferry companies distinguish between high and low seasons. Sports centres offer squash or tennis courts at cheaper rate during the day. Hence, prices are time-dependent in these examples. They are typically summarised in tables with prices associated with a time period. In terms of a relational temporal data model this is a temporal relation.

- Our all-day-life is very often influenced by timetables for buses, trains, flights, university lectures, laboratory access and even cinema, theatre or TV programmes. As one consequence, many people plan their daily activities by using diaries which itself is a kind of timetable. And again: timetables or diaries can be regarded as temporal relations in terms of a relational temporal data model.

- Medical diagnosis often draws conclusions from a patient's history, i.e. from the evolution of his/her illness. The latter is described by a series of values, such as the body temperature, cholesterol concentration in the blood, blood pressure etc. As in the first example, each of these values is only valid during a certain period of time (e.g. a certain day). Typically a doctor would retrieve a patient's values' history, analyse trends and base his diagnosis on his observations.
  Similar examples can be found in many areas that rely on the observation of evolutionary processes, such as environmental studies, economics and many natural sciences.

Temporal Database Management Systems

Temporal database management systems (TDBMS) support the maintenance and manipulation of temporal data in many possible ways. Temporal support can affect many but not necessarily all of the following issues:

1. It can provide an entire *temporal data model* which consists of a temporal

data definition language (DDL) and a temporal data manipulation language (DML). This means that temporal objects can be defined via the DDL and can be created, updated, deleted and retrieved via the DML.

2. User-defined time is already an integral part of the relational data model (time is considered as a domain such as integers or strings). Thus there might be a *temporal query language* that simply offers a set of temporal operators and predicates to enhance the search facilities.

3. Finally, there are various *performance related issues* such as temporal storage structures or the implementation of temporal operators.

We note that 1. and 2. are alternatives that actually depend on the degree of temporal support that one wants to achieve: 1. implies a temporal query language whereas 2. only enhances the very basic temporal facilities given by conventional data models. Any of these two cases will require to be supported by a proper implementation as pointed out in 3.

Temporal-Specific Support

Points 1. and 2. above expose a very variable degree of possible temporal support that can be provided by a database management system (DBMS). Furthermore, we note that a temporal database does not require a TDBMS at all. Temporal databases have existed for many years using conventional[1] DBMS.

These facts are in the centre of a controversy between researchers who support the wide integration of temporal specific features into conventional DBMS and their critics. Davies *et al.*, for example, argue that it is not necessary to provide specific support for temporal data processing but that there are certain general, non-temporal-specific features that have to be incorporated into relational query languages, such as recursion. The latter would not only support temporal features, such as coalescing (see below), but would prove to be useful for many non-temporal situations too [Davies et al., 1995].

One issue of concern is related to the following fact that traditional query languages do not support the many constructs that natural language provides when referring to time or temporal relationships. This not only decreases the user-friendliness of the query language but imposes considerable problems on the query optimiser. Let us look at the following example: Take the sentence "Jack studied at university at the same time as Mark." Using the intervals

[1]By the term *conventional DBMS* we refer to DBMS which do not provide specific support for temporal data processing.

$[j_s, j_e]$ and $[m_s, m_e]$ for representing the respective study start and end dates for Jack and Mark, we can describe the 'same time as' relationship by the expression

$$j_e \geq m_s \wedge m_e \geq j_s \tag{2.1}$$

which pays attention to the fact that 'same time as' does not necessarily mean that Jack and Mark started and finished at the same time but that Jack and Mark were both at university during a certain period of time. Furthermore, it relies on the additional constraints $j_s \leq j_e$ and $m_s \leq m_e$. Alternatively, one could say

$$\begin{aligned} & (m_s \leq j_s \wedge j_s \leq m_e) \\ \vee \quad & (m_s \leq j_e \wedge j_e \leq m_e) \\ \vee \quad & (j_s \leq m_s \wedge m_s \leq j_e) \\ \vee \quad & (j_s \leq m_e \wedge m_e \leq j_e) \end{aligned} \tag{2.2}$$

It should be obvious that neither (2.1) nor (2.2) are straightforward expressions. Things become worse when we consider the fact that most SQL queries are generated automatically by query tools nowadays. Such tools create queries that express the desired query somehow but not necessarily efficiently as this task is left to a query optimiser. Query optimisation, however, is in general a hard problem [Ryan and Smith, 1995]. Although expressions such as (2.2) can theoretically be reduced to (2.1) or to another, less complex expression, it is difficult for an optimiser to recognise and optimise this in practice within a reasonable time frame. In general, if a query tool produces an awful query then there is not much that the optimiser can do about it.

For temporal queries, a possible solution to this problem is to provide temporal operators and predicates that are close to the natural way of expressing the respective relationship. In the case of (2.1) and (2.2) this could be a predicate called 'intersects' which would enable us to say

$$[j_s, j_e] \quad \textit{intersects} \quad [m_s, m_e] \tag{2.3}$$

Expressions, such as this one, not only make the queries less complex and therefore user-friendly but also opens the opportunity to optimise queries semantically:
Conventional DBMSs are tuned to perform well on many standard operations. As seen above, temporal queries are more likely to involve complex constructs like (2.1) or (2.2), e.g. as a join condition. Optimisation techniques can cope

with these to a certain extent but they are likely to result in a poor performance. If temporal specific constructs are provided by the (declarative) query language then more efficient, temporal specific query evaluation strategies can be applied: imagine a query containing (2.3) as a sub-expression. If an optimiser does not know that j_s, j_e and m_s, m_e are the respective start- and endpoints of some timestamp intervals then it does not know about many implicit and and possibly helpful constraints either. Examples for such constraints are:

- A startpoint of a timestamp cannot lie beyond the endpoint, i.e.

$$j_s \leq j_e \text{ and } m_s \leq m_e$$

- Transaction time[2] is restricted to the past and the present. Therefore transaction time timestamps are bound by the current time which is usually referred to as '*now*'. If $[j_s, j_e]$ and $[m_s, m_e]$ are such timestamps then we know that

$$j_s, j_e, m_s, m_e \leq \textit{now}$$

Such implicit conditions can be possibly exploited to increase the performance of query evaluation. To that end, the optimiser must know about the semantics – in this case: *temporal* semantics – of the data.

Among all the arguments for and against temporal-specific support, performance and efficiency of temporal query processing are the least controversial. Many authors have recognised that conventional techniques, such as index structures or join algorithms, are tuned for performing well in standard situations, i.e. atomic data types, equality conditions, etc. These are not necessarily suited to temporal query processing where problems like non-atomic data, temporal predicates, granularity, schema versioning, multiple calendars etc. occur.

In this thesis, we will deal with joins that are based on temporal join conditions, i.e. expressions similar to (2.1) or (2.3). We will show that providing specific techniques for temporal join evaluation is much more efficient than using conventional mechanisms.

Research on Temporal Databases

In the 1980s, observations, like the ones described above, triggered a large number of research efforts on the development of temporal database systems.

[2] See section 2.3 for a description of this concept.

Most researchers concentrated on extending the relational model with temporal features. Some selected examples are HQuel [Tansel, 1986], TQuel [Snodgrass, 1987], the temporal features of Postgres [Stonebraker, 1987], [Stonebraker et al., 1990], DM / T [Jensen et al., 1991], TempSQL [Gadia, 1992] or IXSQL [Lorentzos and Mitsopoulos, 1997]. An impression and overview of temporal database research can be obtained from the temporal database bibliographies that have been published regularly in the SIGMOD Record. The latest two were presented in 1993 [Kline, 1993] and 1996 [Tsotras and Kumar, 1996].

Many research efforts were brought together when a group of researchers discussed a temporal query language, called TSQL2 [Snodgrass et al., 1994], [Snodgrass, 1995], which was based on the SQL92 standard [ISO92, 1992]. The TSQL2 design process tried to integrate many of the features that had been proposed previously. Temporal database researchers met at two workshops, [Pissinou et al., 1994] and [Clifford and Tuzhilin, 1995], and published a book on temporal databases [Tansel et al., 1993]. Since then, temporal databases have become a major topic of interest in almost every database conference.

Currently, the ANSI and ISO committees that are creating the new SQL3 standard are considering a temporal extension of SQL3 which is referred to as *SQL/Temporal* [Darwen, 1997], [Snodgrass, 1996].

2.2 Basic Concepts and Notations

We now want to define some basic concepts and technical terms that are used in the context of *relational* temporal databases. We thereby restrict ourselves to the concepts that are relevant for the remainder of this thesis. We adopt the definitions that have been published by the group of researchers during the process of designing TSQL2, e.g. in [Jensen et al., 1994a]. *This, however, does not imply that any of the work that is presented here and in the remainder of this thesis is specific to TSQL2.* We use these definitions because they can be regarded as being well established among the temporal database research community for the following reasons:

- A large number of researchers were involved in the discussions and did agree upon these terms.
- Many definitions evolved from unifying previously suggested concepts. See [Jensen et al., 1992], [Jensen et al., 1994a], [Jensen et al., 1994b] or [Jensen et al., 1994c], for example. A summary, including many parts of these texts, can be found in [Snodgrass, 1995].

As motivated in the introduction, temporal databases store time-dependent data. In the context of the relational data model this means that temporal data objects – these can be either tuples or single attribute values – have an associated *timestamp* which is a time value, such as a date or a time interval. The most frequently suggested combination – and the one we adopt – is to have temporal relations with timestamped tuples. The advantage of this choice is that this goes well with conventional relational structures: a tuple-timestamp can be regarded as 'just another attribute', at least in some aspects[3]. Temporal relations can even adopt first normal form (1NF) on which many commercial database management systems rely. Alternative approaches, such as timestamping attribute values as in Gadia's *Homogeneous Relational Model* [Gadia, 1988], may not be capable of directly using existing relational query evaluation techniques or storage structures which depend on atomic attribute values. Consequently, many new evaluation techniques would be required and would have to be implemented, always bearing in mind that conventional query evaluation performance should not be penalised in the redesigning process. Given these problems and the fact that radical changes in well established implementations are highly unlikely, it is more realistic to discard such non-1NF approaches.

Figure 2.1 shows a temporal relation `Staff` that is supposed to hold members of a university department, the numbers of their respective offices[4] and a timestamp that indicates the time period in which they worked in the department. A special identifier 'now' is used to denote the current moment. The treatment of 'now' is a separate research topic, see e.g. [Clifford et al., 1997]. For our purposes, we imagine that 'now' is replaced by the current date whenever an operation looks at the data.

Name	Office	Start Date	End Date
Alex	A	1 Apr 92	31 Dec 95
Elisabeth	B	1 Jan 92	now
Frank	A	1 Jan 93	30 Apr 96
Henry	C	10 Jun 90	31 Dec 95
Mary	B	15 Aug 94	now
Vicky	D	10 Jun 90	now

Figure 2.1: Example of a temporal relation `Staff`.

[3]There are critics who argue that time is just another attribute in all respects. The reader might look at [Davies et al., 1995].

[4]For simplicity, we assume that these are the current offices for persons who are still working in the department and the last occupied offices for persons who have already left.

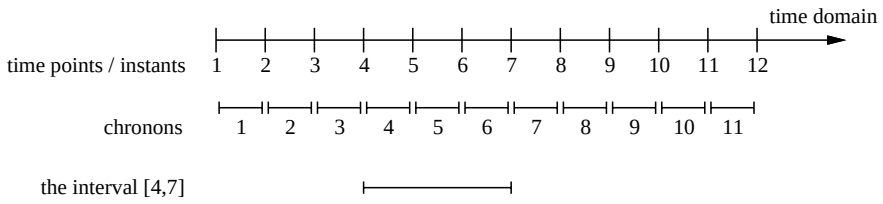


Figure 2.2: Relationship between time domain, timepoints, chronons and intervals.

In general, timestamps of a temporal relation are defined over a certain *time domain* which is often represented as a *time line*. Elements of the time domain are *timepoints* or *instants*. Although time itself is generally perceived to be continuous, most temporal data models that have been proposed are based on a discrete model of time. Such models use a non-decomposable time interval, called a *chronon*, as a basic unit of minimal duration. Starting with an initial time point, following timepoints appear at the distance of a chronon from its predecessor. An *interval* is the time between two timepoints, a start- and an endpoint. Alternatively, it can be interpreted as a contiguous set of chronons.

Figure 2.2 illustrates the relationship between the time domain, timepoints, chronons and intervals. It uses integers to refer to time. This not only simplifies the notation but also avoids the problem of incorporating into our examples the *granularity* of the time line, i.e. the duration of a chronon: a second, a minute, a day etc. This depends on the actual application. As an example, see figure 2.3 which shows the relation `Staff` using an integer time representation, assuming that *now* is at timepoint 10. Below we will describe further details of the choices we make.

Name	Office	Start Date	End Date
Alex	A	3	8
Elisabeth	B	2	10
Frank	A	4	9
Henry	C	1	8
Mary	B	6	10
Vicky	D	1	10

Figure 2.3: Temporal relation `Staff` using an integer time representation.

As already mentioned, we adopt a discrete time domain. This choice does not affect the concepts that are developed and discussed in this thesis but simplifies many notations and discussions. Apart from that, one can find several

practical arguments for the preference of a discrete over a continuous model [Jensen et al., 1994b]: firstly, clocks usually show time in terms of chronons – usually seconds or minutes. Secondly, time references in natural language are normally compatible with the discrete model. Thirdly, the concepts of chronon and interval allow us to naturally model events that are not instantaneous but have a duration. Finally, any implementation of a temporal data model must necessarily have some discrete encoding for time.

As indicated above, a timestamp can be a date, an event or an interval. We adopt the most frequent choice and use interval timestamps. Intervals have proved to be the most versatile representation of time: intervals and relationships between intervals can adequately express almost any time reference in natural language. For that reason, they have been used not only in many temporal database applications but also for many techniques in natural language processing [Allen, 1983]. We usually represent intervals by referring to their start- and endpoint. In the special case that those points are identical the interval has a duration of 0 chronons and therefore depicts a time instant (timepoint). Otherwise the interval has a duration greater than 0 and refers to a contiguous time period[5].

In notational terms, we denote an interval by squared brackets surrounding the respective start- and endpoint:

$$[t_s, t_e] \;=\; \{x : t_s \leq x \leq t_e\} \tag{2.4}$$

or in terms of chronons and if the chronon between timepoints t and $t+1$ is referred to by $\hat{t}$:

$$[t_s, t_e] \;=\; \{\hat{x} : \widehat{t_s} \leq \hat{x} < \widehat{t_e}\} \;=\; \{\widehat{t_s}, \widehat{t_s+1}, \ldots, \widehat{t_e-1}\}$$

See figure 2.2 for an example of an interval $[4, 7]$. We will mainly use the notation used in (2.4).

$[t_s, t_e]$ is also called a *closed* interval. Sometimes it is convenient to exclude the start or the endpoint or both. Such intervals are said to be *left-open*, *right-open* or *open*, respectively, and are denoted by

$$\begin{aligned} (t_s, t_e] &= \{x : t_s < x \leq t_e\} \\ [t_s, t_e) &= \{x : t_s \leq x < t_e\} \\ (t_s, t_e) &= \{x : t_s < x < t_e\} \end{aligned} \tag{2.5}$$

[5]For our purposes, the definitions of a *period* and an *interval* are identical. TSQL2, for example, uses the term *period* to refer to our notion of interval because the term *interval* has already been used in SQL92 for a different concept.

As stated above, we will use integers to represent the time domain. The size of a chronon is one unit, the distance between an integer t and its successor $t+1$. The predecessor of t is referred to as $t-1$. These notations simplify the definitions in (2.5) to

$$\begin{aligned}(t_s, t_e] &= [t_s + 1, t_e] \\ [t_s, t_e) &= [t_s, t_e - 1] \\ (t_s, t_e) &= [t_s + 1, t_e - 1]\end{aligned}$$

We will mainly use the $[t_s, t_e]$ type and use the others whenever it helps to simplify the notation.

For our purposes, we assume that each tuple r of a temporal relation R has at least one interval timestamp. If there is more than one timestamp per tuple then one of them is regarded as the designated one, e.g. the one that is used in a join condition or the one that is used for partitioning the data; the others are treated as conventional attributes.

In summary: each $r \in R$ has an interval timestamp. The startpoint of the timestamp is referred to as $r.t_s$ and the endpoint as $r.t_e$, i.e. the timestamp is the interval $[r.t_s, r.t_e]$. Further notations will be introduced in the stages in which they are required.

2.3 Temporal and Conventional Databases

As outlined in the previous two sections, a temporal database can be regarded as an enhancement of a conventional database: it 'only' sets the data into a context of time, i.e. it adds a time dimension. In a relational database environment, a relation can be considered as a table with rows representing individual tuples and columns holding the respective attribute values. Over time, such a table is updated, i.e. new rows are inserted, some rows are deleted and some attribute values might be modified. This means that the data in the table changes over time. If a copy of the table was taken each time before it is updated and if the date of the copy was added to all rows we could actually follow the evolution of the table. And in fact, this is what many users require: just recall the share-prices-example in section 2.1. In contrast, in many databases, one would only keep the current copy of a table or – as it is frequently called – the current *snapshot*. They are therefore referred to as *snapshot databases*.

Brooks was the first person to elaborate this comparison by regarding the series of snapshots / table copies as a *time cube* [Brooks, 1956]: a snapshot is

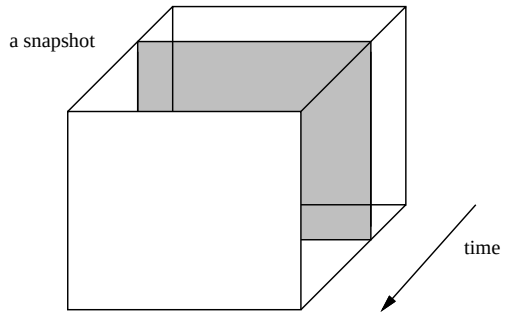


Figure 2.4: A temporal relation as a time cube with a snapshot being a time slice.

then a time slice of such a cube. See figure 2.4. In terms of this analogy, a conventional database always holds one slice whereas a temporal database holds the entire cube. Without going into implementational details, one can imagine that – whenever an update occurs – a conventional database *physically* updates, i.e. throws old values away and stores the new ones, whereas a temporal database is updated *logically*, i.e. it marks the old and new values with timestamps that indicate to which snapshot the actually belong: the current or a previous, historic one. This is also called the concept of *physical* vs. *logical deletion*.

What we have described so far is a time dimension that represented the time dimension from the DBMS's point of view, i.e. the time when (update) transactions in the database take place. This type of time is therefore called *transaction time*. A transaction timestamp indicates the time when the associated tuple is current in the database. Naturally, the most recent value of transaction time is always the current moment, e.g. represented as *now* in figure 2.1. In terms of the time cube, this means that the current snapshot is the front one if time runs from the background to the foreground as in figure 2.4.

There is, however, a second notion of time which is called *valid time*. Whereas transaction time is restricted to the present and the past, valid time extends to the future as well. Imagine a hotel reservation system that stores room bookings in a table. For the staff that run the hotel, it is not really important to know when a room was booked, i.e. when the booking transaction took place (transaction time), but for which time a room is booked (valid time) which naturally must cover the future. Additionally, past bookings might be stored as well, as the management might want to analyse this information in order to analyse customer characteristics.

In the case of valid time the *current* snapshot – if it is supposed to be the

snapshot giving, for example, the current hotel room allocation – might be a slice in the time cube such as the one shown in figure 2.4. As valid time extends to the future it is not necessarily the front one, however.

Whereas a snapshot relation can be considered as a slice of a transaction time cube, a similar connection cannot be drawn for a valid time cube. The contents of a valid time relation that is currently held in the database cannot be regarded as slice. It might be the entire data represented by the cube or parts of it. Remember the example of the hotel reservation system: it suggests that many conventional databases would store a significant amount of the data represented by the valid time cube. This means that the latter are actually valid time databases which treat valid time just as any other attribute. This underlines again that temporal DBMSs emphasise and efficiently support the time dimension(s) but do not extend the expressive power of conventional databases. This is the fact on which many critics build their argument. There is, however, no doubt about the existence and widespread usage of and demand for temporal databases.

2.4 Temporal Databases and Data Warehousing

Temporal databases and data warehousing are two separate areas which are strongly related: data warehouses are the commercial products that require temporal database technology. Naturally, most other database products are amenable to temporal database technology, too. Regarding the market perspectives, however, one has to assume that it will be mainly data warehouses that adopt the techniques that have been and that will be developed by temporal database researchers. In this section, we want to elaborate the connection between data warehousing and temporal databases in some more detail.

A *data warehouse* (DW) integrates information from many, possibly heterogeneous, databases into a physically separated database and makes this information available to analysis [Inmon, 1996]. Figure 2.5 illustrates this concept. The purpose of the analysis might be, for example, to provide the management of a company with information on trends and facts that are required for taking strategic decisions.

Trend analysis can go along many dimensions, the most important of which is *time*. It is used to detect certain characteristics in the evolution of data, e.g. over time or over various geographic regions or over product lines. In the case of temporal evolution, this means that a data warehouse is very often required

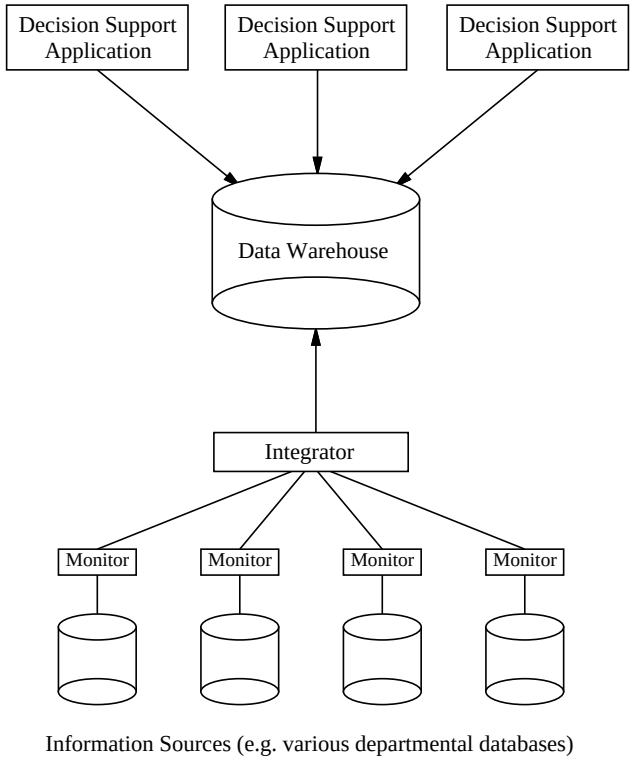


Figure 2.5: The concept of a data warehouse.

not only to hold a reformatted subset of current operational data, e.g. sales figures, but also a history of this data. This is nothing other than a *historical database*, a special case of a temporal database [Sarda, 1993]. For that reason, Inmon says that a "salient characteristic of the data warehouse is that it is time variant". Furthermore he comments:

- Data warehouses are required to hold data of the last 5 to 10 years whereas operational databases[6] require a 60 to 90 days time horizon.
- "Operational databases contain current value data – data whose accuracy is valid as of the moment of access. As such, current value data can be updated. Data Warehouse data is nothing more than a sophisticated series of snapshots[7], taken at one moment in time." This comment corresponds widely with the concepts of the time cube and physical vs. logical deletion that were introduced in section 2.3.

[6] An operational database manages the data that is required for day-to-day operations of a company, such as reservation systems in the case of a travel company. This stands in contrast to data warehouses whose purpose is to provide information to support decision-taking in the management of a company, such as customer behaviour and market trends.

[7] We note that this is a gross over-simplification. The data warehouse, for example, must also take into account changes in the schemata and in the semantics of the data over time.

- "The key structure of operational data may or may not contain some element of time [. . .] The key structure of the data warehouse always contains some element of time."

These comments imply that data warehousing is a discipline that adopts temporal database concepts among many others. And in fact, many references to temporal database functionality can be found by data warehouse vendors:

- Many data warehouse vendors claim that their products are capable of processing *historical data / information*. Examples of such vendors are RedBrick [Red Brick Systems, 1995a], Informix [Informix Inc., 1995], Prism Solutions [Prism Solutions Inc., 1996], Oracle [Oracle Corp., 1996].

- Researchers from SAP claim that data warehouses must have the ability to meaningfully link and cross-reference data "applying *time-related criteria*". Furthermore they state that one salient feature of DW data management is the "*time-variant* data organization" [Heinrich and Hofmann, 1996].

- Red Brick Systems call their product *RedBrick Warehouse VPT*, in which 'T' stands for the fact that it provides "*time-based* data management" [Red Brick Systems, 1995c].

Data warehousing is widely regarded as a discipline which has been taken over by industry. And actually until recently, there were only very few academic research groups looking at data warehouses. Many critics call it a buzz word that has been bent by many marketing departments in order to position products in a market with a thriving prospect [International Data Corporation (IDC), 1996]. Across the board it is probably fair to say that the term *data warehouse* is stamped by industry nowadays. In contrast to that, there are *temporal databases* as one of many (academic) disciplines that have an impact on data warehouse products. Hence, whenever we speak about the practical or commercial application of temporal database technology we have to keep data warehousing applications in mind.

Chapter 3

Join Processing

In this chapter, we introduce the basic ideas that stand behind join processing in conventional relational database systems. Understanding the principles, techniques and experiences of traditional join processing is a precondition for understanding (a) the different efficiency considerations that are imposed by temporal joins, and (b) the decisions that we take when designing efficient temporal join techniques in the oncoming chapters.

Section 3.1 formally defines the join operation. In section 3.2, the role of the join operation in the relational data model is elaborated. This should make the reader aware of the significance and the importance of the join and efficient join processing. In section 3.3, we introduce some types of joins. Traditionally, both the vendors and the research community have mainly focused on one type of join, namely the *equi-join*, because it is by far the most frequent one in typical database installations. Nevertheless, it is stressed that there is a rising need for specific types of joins, such as temporal or spatial joins. Section 3.4 presents a wide range of sequential join algorithms. Although the emphasis is put on equi-joins we also discuss issues regarding nonequi-joins. Similarly, section 3.5 presents techniques for processing joins in parallel. Finally, a classification schema for join algorithms is given in section 3.6.

3.1 Definition of the Join

The join operation, denoted by $\bowtie$, combines two relations R and Q to form a new relation S. If the attributes of R are referred to as $A_1, A_2, \ldots, A_m$ and those of Q as $B_1, B_2, \ldots, B_n$ then S has the $m + n$ attributes $A_1, \ldots, A_m, B_1, \ldots B_n$. Tuples s of S are formed by concatenating a tuple $r \in R$ with a tuple $q \in Q$, denoted as $s = r \circ q$. Usually there is a *join condition* C that has to be fulfilled by the tuples r and q that are concatenated to form an s. In total, the notation

for the join looks like this:

$$S \;=\; R \bowtie_C Q$$

The most frequently used condition is that an r has to hold the same value in a certain attribute, say A_i, as a q in an attribute, say B_j, if they are to be concatenated to form an s. This join condition is denoted as $R.A_i = Q.B_j$, and $R.A_i$ and $Q.B_j$ are said to be the *join attributes*.

Put differently: the join $R \bowtie_C Q$ is a subset of the *cartesian product* $R \times Q$[1]. The cartesian product of R and Q concatenates each tuple of R with every tuple of Q. This results in a new relation $R \times Q$ with $|R| \cdot |Q|$ tuples, with $|R|$ and $|Q|$ being the cardinalities of R and Q respectively. The result of the join $R \bowtie_C Q$ can then be retrieved from $R \times Q$ by selection over the join condition C. Thus

$$R \bowtie_C Q \;=\; \sigma_C(R \times Q) \tag{3.1}$$

As an example for a join, consider the two relations `Staff` and `Student` of figure 3.1. Imagine that these relations respectively hold members of staff and students of a certain university department. Staff members are described by their name, office, start and end dates of the period they worked in the department. Similarly, students have a name, a workroom that is assigned to them, a start and an end date. For simplicity, we assume that names are unique.

Name	Office	Start	End
Alex	A	3	8
Elisabeth	B	2	10
Frank	A	4	9
Henry	C	1	8
Mary	B	6	10
Vicky	D	1	10

(a) `Staff`

Name	Workroom	Start	End
Charles	X	1	4
Frank	Y	1	4
Karen	Y	7	9
Mary	Y	2	5
Olga	Z	1	3
Steve	Z	7	10

(b) `Student`

Figure 3.1: Example relations holding staff members and students.

A typical query would be to find staff-student pairs who started in the department at the same time. These can be found by a join

$$\texttt{Staff} \bowtie_C \texttt{Student}$$

[1]In strict terms this is not true because the cartesian product results in tuple pairs (r, q) whereas the join creates tuples that origin in concatenations $r \circ q$ of tuples. This formality, however, is usually ignored by many authors. For our purposes, it can be ignored too.

using the join condition

$$C \equiv \texttt{Staff.Start} = \texttt{Student.Start}$$

Figure 3.2 shows the result.

Name	Office	Start	End	Name	Workroom	Start	End
Elisabeth	B	2	10	Mary	Y	2	5
Henry	C	1	8	Charles	X	1	4
Henry	C	1	8	Frank	Y	1	4
Henry	C	1	8	Olga	Z	1	3
Vicky	D	1	10	Charles	X	1	4
Vicky	D	1	10	Frank	Y	1	4
Vicky	D	1	10	Olga	Z	1	3

Figure 3.2: Result of the join $\texttt{Staff} \bowtie_C \texttt{Student}$.

3.2 Role of the Join Operation

The join operation is one of the most investigated research issues in the context of the relational data model. For over two decades, numerous papers have been published on join-related topics. Alone the overview paper by Mishra and Eich [Mishra and Eich, 1992] refers to 198 join-related publications and there is a huge number of papers that were not mentioned and that have been published since. Join processing has been studied from many different points of view, such as query optimisation, I/O optimisation, buffer usage optimisation, hardware support, parallel processing or physical database design. There are two major reasons why the join has attracted that much attention:

- it is frequently used, and
- it is one of the most costly and most data-intensive relational operations.

Both aspects are discussed in the following two subsections.

3.2.1 The Significance of the Join in Relation Query Processing

For understanding the significance of the join for relational query processing we want to see why and where the necessity of a join arises. Amongst many reasons, it is already the database design process which creates an oncoming demand for joins within relational queries. Let us go one step back from the staff-student scenario as it was described in section 3.1: imagine that someone designs a database for this department. An initial step would be to create a conceptual schema in some semantic data model, e.g. the entity-relationship (ER) model [Chen, 1976]. A conceptual schema is then mapped into a logical schema using one of various logical data models, such as the network, the relational or the object-oriented model[2]. We want to look at one aspect of this mapping in more detail in order to identify one reason for which the join operation is so important in relational query processing.

Figure 3.3 shows part of the departmental scenario in entity-relationship notation [Korth and Silberschatz, 1991]. There are two entity sets, namely `Staff` and `Student`, which are related via two relationships, namely `teaches` and `supervises`. It is intended to describe a department that has a number of staff and a number of students. Staff members teach students in

[2]See [CODASYL, 1971], [Codd, 1970] and [Cattell, 1996] for details of these models.

various courses and also supervise students in certain research projects. For simplicity, courses and projects are omitted.

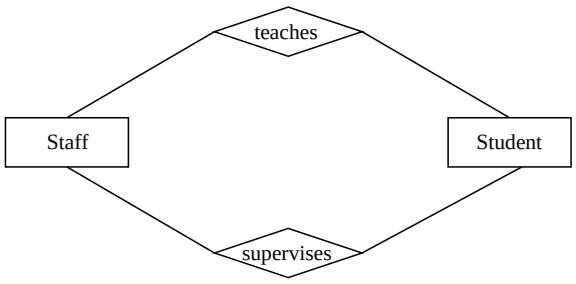


Figure 3.3: An example of a conceptual database design in entity-relationship notation.

This conceptual schema can now be translated into a logical schema. Depending on the logical data model that we choose, we will get different results. For our purpose, we want to look at the way in which the relationships between entities are translated, such as the `teaches` relationship between `Staff` and `Student`.

In the relational model, tuples within a relation are identified by a unique attribute value or a unique combination of the latter. This is a called a *key*. The `Name` attributes in `Staff` and `Student` are examples for keys as we assumed names to be unique. Keys are a kind of *logical* reference as opposed to a *physical* reference, such as a memory address. In our example, one can map the `teaches` relationship set of figure 3.3, the conceptual schema, into a relation `Teaches` in the relational schema. The relation `Teaches` holds tuples which consist of two parts: a key that refers to a tuple in the `Staff` relation and a key referring to a tuple in the `Student` relation. See figure 3.4. If a tuple's key is used within another relation for such purposes then the corresponding attribute in the other relation is called a *foreign key*. Thus a link between tuples of two different relations is established in a rather abstract way: it can be deduced from the condition '*key = foreign key*'.

One might argue that the conceptual separation of relations does not have to translate to the physical level, i.e. data could be physically linked although being conceptually separated. This, however, is only sometimes the case. Relations are usually normalised because this facilitates updates and helps to maintain a consistent database.

Whereas relational database systems are based on such logical links, systems that are based on the network, such as ADABAS [Tsichritzis and Lo-

Teacher	Student
Alex	Charles
Alex	Frank
Alex	Mary
Elisabeth	Charles
Elisabeth	Karen
...	...

Figure 3.4: Relation `Teaches`.

chovsky, 1977], or some systems with an object-oriented data model, such as ObjectStore [Lamb et al., 1991], would create *physical* links between the entities: they would, for example, use pointers to link a staff member with a student if there exists a 'teaches' relationship between them.

The operation that creates physical links between logically linked tuples of two (or more) relations is the *join*. The most frequent usage – as motivated in the previous paragraphs – is to materialise *'key = foreign key'* relationships. These arise from mapping ER-like relationship sets, such as `teaches` in figure 3.3, to relations in a relational database. Estimates are that around 90% of join conditions are such *'key = foreign key'* conditions [Valduriez, 1987].

As an example, consider the two *'key = foreign key'* relationships between the relations of figures 3.1 and 3.4, namely `Staff.Name = Teaches.Teacher` between the `Staff` and `Teaches` relations and `Student.Name = Teaches.Student` between the `Student` and `Teaches` relations. If, for example, we were searching for the workrooms of Alex's students we would need to materialise the *'key = foreign key'* link between the `Student` and the `Teaches` relations. This can be done by *joining* the two relations, using `Student.Name = Teaches.Student` as the *join condition* (see figure 3.5 (a)), followed by a selection of Alex's students and a projection on the `workroom` attribute (see figure 3.5 (b)).

The join operation, however, is not restricted to *'key = foreign key'* but has many other applications. As an example of a non-*'key = foreign key'*-link, we might want to find staff members who share an office in our scenario. To that end, the relation `Staff` can be joined with itself using `Staff A.Office = Staff B.Office` as the join condition. Another example would be to look for students who became a staff member by joining the `Staff` and `Student` relations via the `Staff.Name = Student.Name` condition[3].

Up to here, all examples of joins used equality as the predicate in the join

[3]Please remember that, in the scenario, names are assumed to be unique for simplicity.

Name	Workroom	Start	End	Teacher	Student
Charles	X	1	4	Alex	Charles
Frank	Y	1	4	Alex	Frank
Mary	Y	2	5	Alex	Mary
Charles	X	1	4	Elisabeth	Charles
Karen	Y	7	9	Elisabeth	Karen
...	...	...	...	...	...

(a) Result of the join `Student` $\bowtie_C$ `Teaches` with $C \equiv$ `Student.Name = Teaches.Student`.

Workroom
X
Y

(b) Final result of $\pi_{\texttt{workroom}}(\sigma_{\texttt{Teacher='Alex'}}(\texttt{Student} \bowtie_C \texttt{Teaches}))$.

Figure 3.5: Example of a *'key = foreign key'* join.

condition. Such joins are called *equi-joins*. An, admittedly artificial, example of a *nonequi-join* is to find staff-student pairs in which the staff member worked in the department before the respective student entered. Such pairs can be found by joining `Staff` and `Student` using `Staff.End` $<$ `Student.Start` as the condition. This is actually an example of a *temporal join* as the join condition is based on a temporal relationship between the timestamps of two temporal relations. More precisely, it is a *before join* because 'before' is the temporal relationship. Join types, such as equi- and nonequi-joins, are discussed in some more detail in section 3.3 and, for the temporal case, in chapter 4.

In summary, we have seen that the join operation is used to materialise the various logical links that exist between data of two or more relations. Whenever a query needs to relate data of two or more relations a join operation is required. This situation appears frequently because of the many *'key = foreign key'* relationships that are introduced in the database design process, e.g. when translating relationship sets of a entity-relationship model into relations of the relational data model or through normalisation of relations. Furthermore, there are many more logical links of various types that might be materialised by a join operation.

3.2.2 Join Performance Issues

The join operation is closely related to the cartesian product as we have seen in (3.1). This means that, initially, each tuple of the cartesian product must be considered for the join result. The size of the cartesian product $|R| \cdot |Q|$, however, is huge in comparison with the sizes of the participating relations, $|R|$ and $|Q|$, with the latter already being large in many cases. This can involve a huge number of disk accesses to retrieve tuples of R and Q which implies a very poor performance.

The enormous amount of data and data movement and very poor performances for naive approaches have triggered a lot of research efforts aiming for improvement. The latter are summarised in the remainder of this chapter.

3.3 Types of Joins

The join condition plays a very important role in join processing. Certain types of conditions allow certain processing and optimisation techniques. For that purpose, it has proved to be very useful to classify joins depending on the respective join condition. This section summarises the most prominent types in traditional relational queries. Please note that these categories are not disjoint but emphasise certain features of the join condition, such as the data types of the join attributes or the predicates that are involved.

Theta-Joins: Many join conditions are based on the pattern

$$R.A \ \theta \ Q.B \tag{3.2}$$

where θ is one of the following predicates / operators: $=$, $\neq$, $<$, $>$, $\leq$ or $\geq$. Joins with such a condition are called *theta-joins*. Naturally $R.A$ and $Q.B$ must be attributes that are comparable by these θ operators; it would not make sense to have strings in $R.A$ and integers in $Q.B$ and compare them by $\leq$. More generally, theta-conditions consist of various simple ones of the form described in (3.2) which are connected by logical operator, such as AND and OR.

Equi-Joins: An *equi-join* has a join condition that is based on the equality predicate $=$. It therefore is a special case of a theta-join. The majority of join conditions are believed to be based on an equality predicate, such as in *'key = foreign key'* conditions. Consequently, most join algorithms have been optimised for equi-conditions. See section 3.4 for details.

Nonequi-Joins: Theta-joins which are not equi-joins are called *nonequi-joins* [Mishra and Eich, 1992]. Nonequi-joins have attracted less research efforts than equi-joins due to their rare usage. However, new data types, such as timestamps, intervals, rectangles, polygons etc., have become more interesting with the growing requirements and ambitions of data modeling. In the context of relational query processing this means that there are many possible new join conditions that describe relationships between objects of one of these data types. These relationships very often can be described in the form of a nonequi-join condition. This has triggered some interest in new types of data-type-specific joins, such as those described in the following two paragraphs. Most of them can be considered as traditional nonequi-joins.

Temporal Joins: This class of joins involves temporal data types, such as time intervals or dates. The interval type is predominant in most temporal data models. Therefore, temporal join conditions essentially describe relationships between intervals. In the context of time representation in natural language processing, Allen identified 13 possible relationships between two intervals [Allen, 1983]. Each of them implies its own subtype of temporal join. As an example, you might refer to the *before join* presented in section 3.2.1. Temporal joins are discussed in detail in the next chapter.

Spatial Joins: Spatial data types, such as rectangles or polygons, are used in geographic information systems (GIS). Imagine two spatial relations, one holding areas (polygons) with nuclear plants and the other one storing areas with high cancer rates. If we wanted to investigate possible spatial connections between nuclear plants and cancer rates we would need to join these two relations using 'intersection of areas' as the (spatial) join condition. Spatial joins are more dynamic than temporal joins in the sense that many geometric data types have to be catered for rather than one or two. On one hand, this makes them more general but, on the other hand, allows hardly any data-type-specific optimisation. For further details on spatial joins the reader might refer to [Günther, 1993], [Lo and Ravishankar, 1996] or [Patel and DeWitt, 1996], for example.

3.4 Sequential Join Algorithms

As previously stated, a lot of effort has been spent on finding efficient ways of joining two or more relations. As a result, many different join algorithms have evolved, many of them being variations of others. Essentially, there are four basic techniques:

- brute force nested-loops,
- sort-merge,
- hash,
- data-structure-assisted (index-based).

In the following subsections, we look at these techniques for performing the join

$$R \bowtie_C Q$$

The purpose is to give sufficient details to understand the implications for temporal joins. A more detailed summary of join algorithms can be found in [Mishra and Eich, 1992].

Finally, in section 3.6, two features are presented that allow us to classify the various algorithms, namely:

- the way in which data is partitioned into data fragments, and
- the degree of overlap of fragments during the joining phase.

These characteristics will allow us to analyse the algorithms with respect to their suitability for temporal join processing.

3.4.1 Brute Force Nested-Loops Joins

This is the simplest join technique. It is based on equation (3.1). One of the relations being joined, say R, is designated as the *outer relation*, and the other one, Q, as the *inner relation*. From the correctness point of view, it does not matter which of the two participating relations is the outer and which is the inner relation.

For each tuple r of the outer relation, all tuples q of the inner relation are read and compared with the tuple from the outer relation. Whenever the join condition C is satisfied, the two tuples are concatenated to form a tuple $r \circ q$ which is placed in an output relation.

In other words: each tuple of the cartesian product $R \times Q$ is tested on the join condition C. If it satisfies C then it is included into the join result. Figure 3.6 summarises the algorithm. Figure 3.7 gives a visual example of how the algorithm finds the join result: tuples of R are scattered along the horizontal axis; the respective value of the join attribute is put below each of the resulting columns. Similarly, the tuples of Q are put along the vertical axis. The resulting grid represents every possible comparison between tuples of R and Q and as such the search space of the join, i.e. the cartesian product $R \times Q$. Comparisons that satisfy the join condition, i.e. tuple combinations that contribute to the join result, are shown in dark grey whereas unsuccessful comparisons are put in light grey. The figure shows that the brute force nested-loops join performs an exhaustive search over the cartesian product.

```
for each tuple r ∈ R do
    for each tuple q ∈ Q do
        if r and q satisfy C then
            put r ∘ q in the output relation
        fi
    od
od
```

Figure 3.6: Brute force nested-loops join.

The brute force nested-loops join is frequently referred to simply as a *nested-loop join*. C.J. Date, however, pointed out that this is misleading as all the other join techniques also use nested loops in one or the other way [Date, 1995]. For simplicity and because of the fact, that it has become a commonly accepted expression, we will stick to calling it 'nested-loop join' in the remainder of the thesis.

In practice, the nested-loop join is implemented as a nested-block join [El-Masri and Navathe, 1994]: instead of retrieving one tuple each time, an entire block of tuples is read from disk and cached in main memory. This is more efficient because, this way, several random accesses are replaced by sequential tuple accesses which are faster. A further performance improving measure is to use the relation with the lower cardinality as the outer relation in order to reduce the I/O costs which are given by the formula

$$p_R + p_R \cdot p_Q \tag{3.3}$$

where p_R stands for the size of the outer relation R (in pages) and p_Q for the size

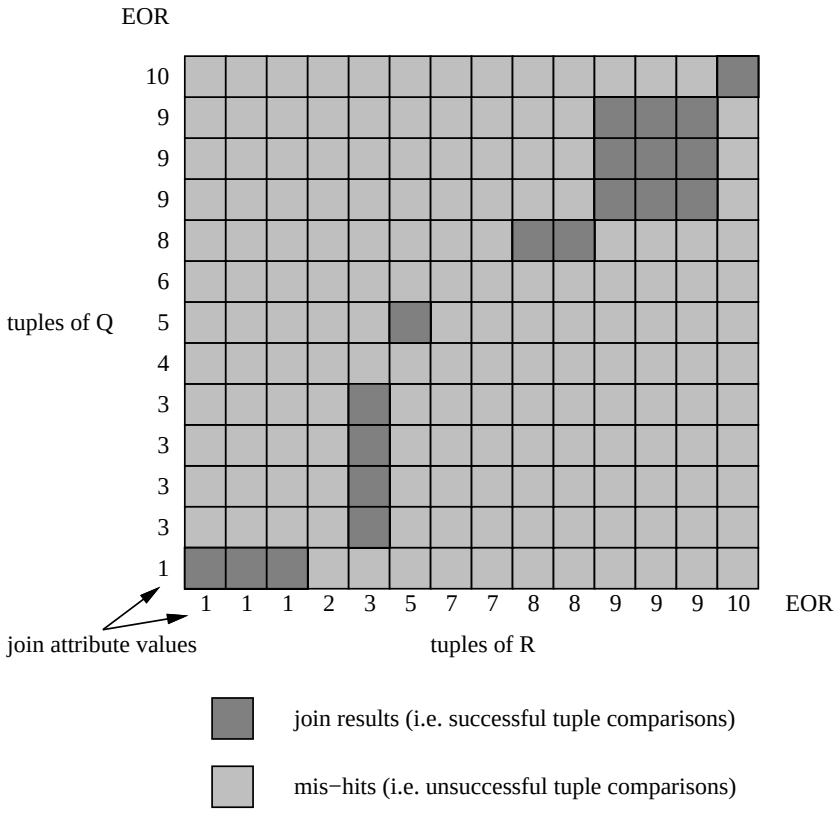


Figure 3.7: Search strategy of the brute force nested-loops join.

of the inner relation Q (in pages). (3.3) is minimised if and $p_R \leq p_Q$ [El-Masri and Navathe, 1994].

Furthermore, we can switch the direction in which the inner relation is read each time. This has the advantage that the last block read by the previous inner loop is the first block of the oncoming inner loop. It still is in a memory buffer and therefore does not need to be read from disk. This method is called *rocking* and was proposed by [Kim, 1980]. Naturally, every method that accelerates disk access to tuples, such as using an index, helps to improve the performance of the algorithm.

The problem of the nested-loop join lies in the exhaustive matching. Whenever the join condition C causes only a small fraction of the cartesian product to be part of the join result the nested-loop technique performs a large quantity of unsuccessful comparisons (see **if**-statement in figure 3.6). Such a situation is characterised by a low *join selectivity* [Piatetsky-Shapiro and Connell, 1984] which is defined as

$$\text{join selectivity} = \frac{\text{size of the join result}}{\text{size of the cartesian product}} = \frac{|R \bowtie_C Q|}{|R| \cdot |Q|} \tag{3.4}$$

Whereas high selectivities suggest that the effort of comparing every tuple of one relation with every tuple in the other is justified, it is the opposite for low selectivities. We use this observation as a motivation for looking at alternative techniques in the following sections.

3.4.2 Sort-Merge Joins

As we have seen from the discussion of the nested-loop join, an exhaustive comparison might not be very efficient in many situations. One possibility to avoid this is the following:

1. Both relations are sorted on the join attributes.
2. Then, both relations are scanned in the order of the join attributes. Tuples that satisfy the join condition are merged to form the result relation.

This technique is called a *sort-merge* join.

The concrete sort-merge join algorithm depends on the actual join condition, in the case of a theta-join, for example, on the θ operator. Furthermore, it will depend on whether join attributes are keys or not. Let us consider the case of an equi-join $R \bowtie_C Q$ with $C \equiv R.A = Q.B$ with A and B being the

```
/* Stage 1: Sorting */
sort R on R.A
sort Q on Q.B

/* Stage 2: Merging */
r = first tuple in R
q = first tuple in Q

while r ≠ EOR and q ≠ EOR do
    if r.A > q.B then
        q = next tuple in Q after q
    else
    if r.A < q.B then
        r = next tuple in R after r
    else
        put r ∘ q in the output relation

        /* output further tuples that match with r */
        q' = next tuple in Q after q
        while q' ≠ EOR and r.A = q'.B do
            put r ∘ q' in the output relation
            q' = next tuple in Q after q'
        od

        /* output further tuples that match with q */
        r' = next tuple in R after r
        while r' ≠ EOR and r'.A = q.B do
            put r' ∘ q in the output relation
            r' = next tuple in R after r'
        od

        r = next tuple in R after r
        q = next tuple in Q after q
    fi
od
```

Remark: *EOR* denotes an 'end-of-relation' mark that is returned by either the 'first tuple in' or the 'next tuple in' operation if it fails to retrieve a tuple because the end of the respective relation has been reached.

Figure 3.8: Sort-merge join algorithm for equi-joins.

integer-valued attributes. The algorithm can then look as shown in figure 3.8. It can be simplified if A or B is a key.

The advantage of the sort-merge equi-join is that each relation, if sorted, is scanned only once, thus we have $|R| + |Q|$ tuple accesses on disk compared to $|R| \cdot |Q|$ for the nested-loop join. Furthermore, the number of unnecessary comparisons is relatively low in any situation. This is very beneficial in the case of a low join selectivity in which many unnecessary comparisons occur when checking all tuples of the cartesian product. This becomes obvious from figure 3.9 which uses the same scenario as figure 3.7 but marks the tuple comparisons performed by the sort-merge equi-join algorithm of figure 3.8. There is only a very small number of mis-hits. The trick behind this strategy is the following: in figure 3.9, the tuples are sorted on the join attribute which implies that the successful comparisons are to be found roughly around the diagonal of the grid (in the case of an equi-join). The sort-merge strategy follows the path along this diagonal and corrects the direction whenever it encounters a mis-hit.

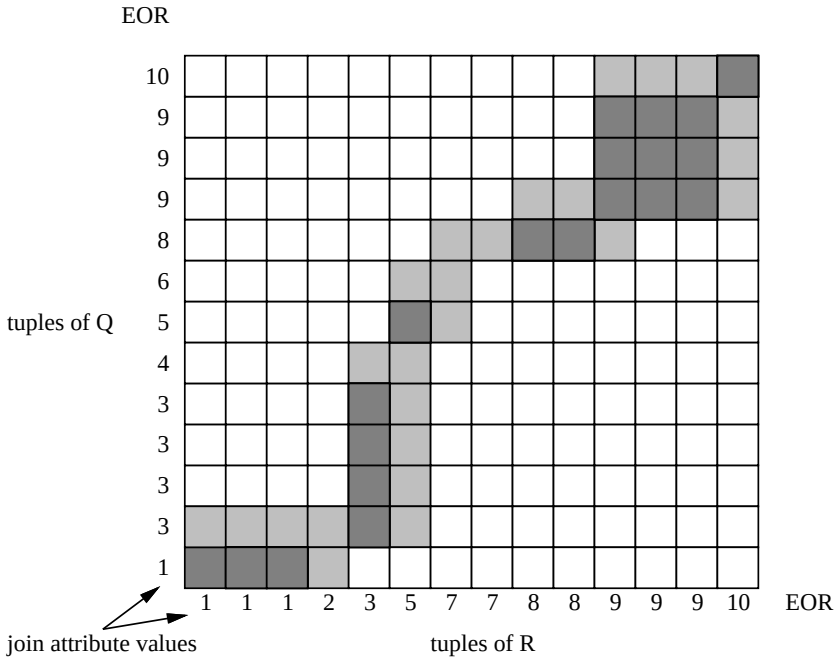


Figure 3.9: Search strategy of a sort-merge equi-join.

The problem of the sort-merge join lies in the requirement that relations have to be sorted on the join attributes prior to the merging stage. In general, this has proved to be the determining component of the execution time

[Mishra and Eich, 1992]. If a join needs to be performed frequently for different queries, then the database administrator can choose to sort the table on the join attribute. Many relations, for example, are sorted on their respective key attribute(s). Thus the sorting stage can be omitted for such a relation if the key attribute(s) is also the join attribute(s).

The sort-merge join is fairly robust and is the best choice in many cases, especially when no indices exist over the join attributes [Blasgen and Eswaran, 1977], [Su, 1988].

3.4.3 Hash Joins

The performance gain achieved by the sort-merge join is based on the facts that during the sorting stage

(1) tuples with the same join attribute value are grouped, and

(2) tuples with similar[4] join attribute values are also nearby.

For an equi-join the sort-merge algorithm only exploits (1). This suggests that sorting the relations is actually too much; methods that create a situation as described by (1) are sufficient.

Hashing the relations is such a method. It creates a set of hash buckets with each bucket being associated either with one join attribute value or a range or set of such values. A hash function h that takes a join attribute value as an argument then assigns a tuple to a bucket. Consider, for example, the following join that was discussed in section 3.2:

$$\texttt{Staff} \bowtie_C \texttt{Student}$$

$$\text{with } C \equiv \texttt{Staff.Name} = \texttt{Student.Name}$$

We can hash the relation `Staff` on the join attribute `Name` using a hash function that assigns the tuples to three buckets. Each bucket is supposed to hold tuples[5] according to the letter with which the `Name` value starts. See figure 3.10 for the result. Next, we can subsequently read tuples from `Student`. Using the same hash function, we can find out in which bucket those tuples of `Staff` are to be found that can match with the respective tuple of `Student`. Put the

[4]The notion of 'similar' depends on the data type; for numeric values this might be a low difference in values whereas for strings it can be same prefixes etc.

[5]or *references to tuples* to reduce the amount of memory required.

other way: all the tuples in the other buckets can be discarded; the actual join for a particular tuple can be concentrated on the tuples in a certain bucket. This algorithm is called the *simple hash join* [DeWitt and Gerber, 1985]. It is summarised in figure 3.11 for the case of an equi-join.

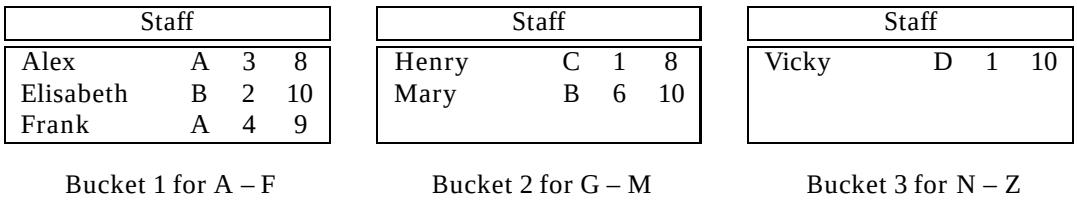


Figure 3.10: Example for hashing the relation `Staff` into buckets.

```
/* Hash relation R */
for each tuple r ∈ R do
    put r in bucket no. h(r.A)
od

/* Probe relation Q */
for each tuple q ∈ Q do
    for each tuple r in bucket no. h(q.A) do
        if r.A = q.B then
            put r ∘ q in the output relation
        fi
    od
od
```

Figure 3.11: Simple hash join.

The search strategy for a simple hash join depends on the respective hash function that is employed. Figures 3.12 and 3.13 show two examples for the scenario that we have already used for analysing the nested-loop and sort-merge algorithms. In this case, the hash function divides the join attributes' domain into several ranges. In figure 3.12, for example, five ranges are created, each one associated with a bucket. The hash join algorithm then searches the 'rectangles' that arise from corresponding buckets. This causes only a small number of mis-hits if the partitioning is not complete (figure 3.12) and none if it is (figure 3.13).

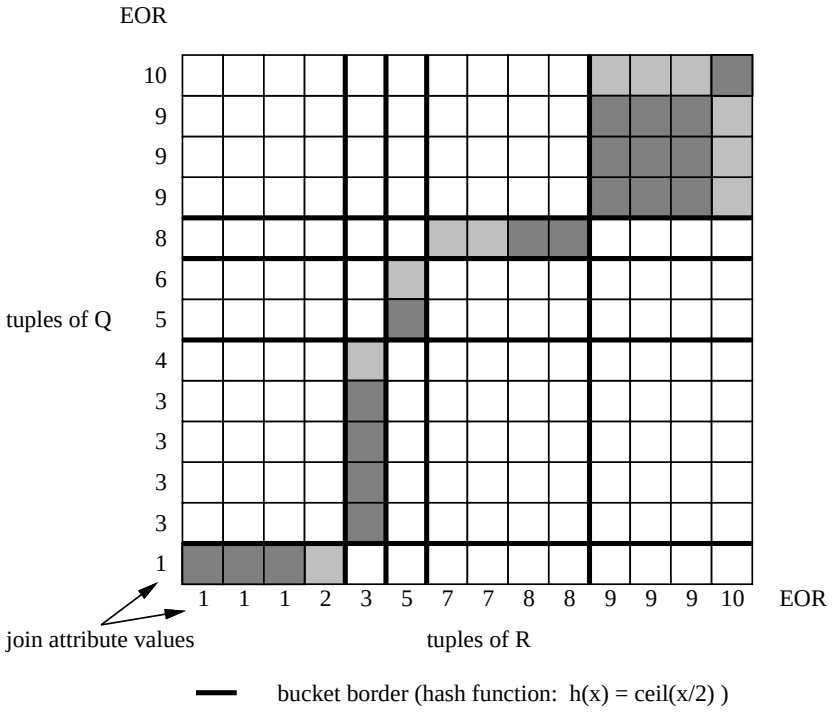


Figure 3.12: Search strategy of a simple hash equi-join.

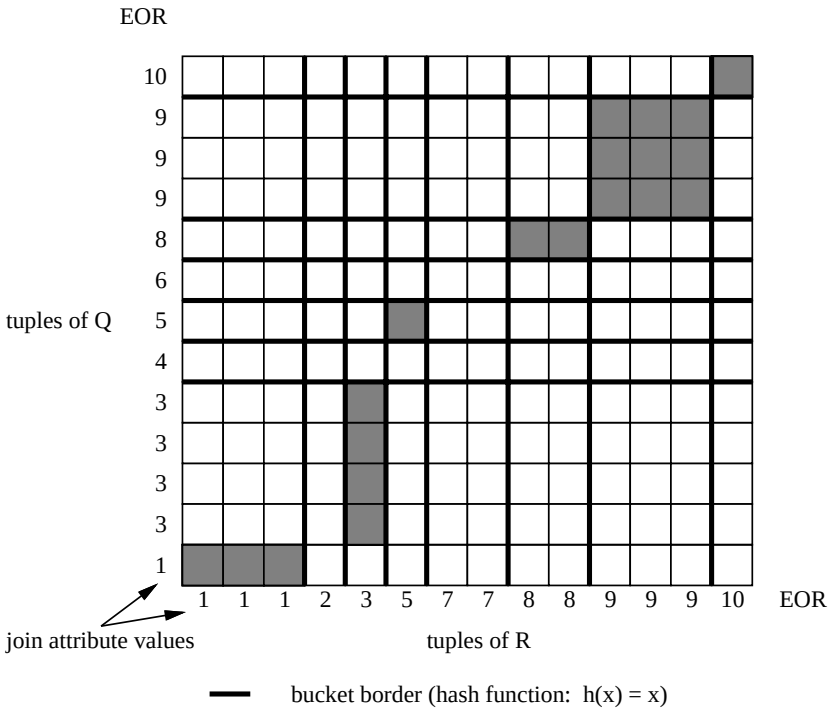


Figure 3.13: Search strategy of a simple hash equi-join with complete partitioning.

Hash joins have several advantages. If the hash table, i.e. the set of all hash buckets, fits entirely into main memory, the hash join has to scan each relation only once. For that reason, it is usually the smaller of the two relations that is hashed in the beginning [Bratbergsengen, 1984]. The performance of the hash join depends on the quality of the hash function and its implications, such as the number of buckets or the value ranges that are assigned to the buckets. If there are only a few ranges then there might be a large number of unnecessary comparisons because each bucket has to hold a large number of tuples. Furthermore, *data skew* [Walton et al., 1991], i.e. non-uniformly distributed data, and an inadequate choice of value ranges can cause certain buckets to hold a large number of values whereas others might be empty. This can also lead to a large number of unnecessary comparisons.

A variation of the simple hash equi-join, as illustrated in figure 3.11, is the *Grace hash equi-join* which was proposed in [Kitsuregawa et al., 1984]. It precedes the simple hash join by an additional partitioning stage: first, relations R and Q are hashed into buckets $R_1, \ldots, R_m$ and $Q_1, \ldots Q_m$ using a hash function h_1. This creates a situation in which tuples of a bucket R_k can only join with tuples in Q_k. Thus the join $R \bowtie_C Q$ is divided into or *partial joins* $R_1 \bowtie_C Q_1$, ... , $R_m \bowtie_C Q_m$ which are performed by a simple hash join each. See figure 3.14. Obviously, the Grace hash join method can be generalised allowing sort-merge or nested-loop techniques for processing the partial joins $R_k \bowtie_C Q_k$. Furthermore, the computation of the partial joins can be concurrent. These aspects will be discussed in section 3.5.

A further improvement of the Grace hash join was proposed in [DeWitt and Gerber, 1985] and [Shapiro, 1986]: instead of flushing tuples of R_1 to disk they are kept in memory and joined with tuples that are found to fall into Q_1 during the hashing stage. Thus the joining stage only deals with the joins $R_2 \bowtie_C Q_2$, ... , $R_m \bowtie_C Q_m$.

In general, hash-based joins have been found to be some of the most efficient join techniques [Gerber, 1986]. Problems arise in situations with heavily skewed data or for nonequi-joins. Hashing is not only useful to reduce the number of unnecessary comparisons but also allows the decomposition of one big join operation into several smaller and independent ones. This is especially important for the parallelisation of joins which is to be discussed in section 3.5.

In many situation, hashing has proved to be more efficient than sorting. The latter observation has motivated G. Graefe to publish a paper with the title *Sort-Merge-Join: An Idea Whose Time Has(h) Passed?* [Graefe, 1994] in which he

```
/* Hash relation R */
for each tuple r ∈ R do
    put r in bucket (output buffer) k = h1(r.A)
od
flush output buffers 1, ..., m to disk

/* Hash relation Q */
for each tuple q ∈ Q do
    put q in bucket (output buffer) k = h1(q.B)
od
flush output buffers 1, ..., m to disk

/* Simple hash join for Rk ⋈C Qk */
for k = 1 to m do
    for each tuple r ∈ Rk do
        put r in bucket no. h2(r.A)
    od
    for each tuple q ∈ Qk do
        for each tuple r in bucket no. h2(q.B) do
            if r.A = q.B then
                put r ∘ q in the output relation
            fi
        od
    od
od
```

Figure 3.14: Grace hash join.

analyses the duality of sort- and hash-based query processing. He concludes that the sort-merge approach (for equi-joins) is almost obsolete with very few exceptions.

3.4.4 Data-Structure-Assisted Joins

The class of data-structure-assisted joins makes use of special data structures which can be regarded as some kind of index. Many such data structures have been proposed, such as join indexes [Valduriez, 1987], Bc-trees [Goyal et al., 1988], T-trees [Lehman and Carey, 1986], kd-trees [Kitsuregawa et al., 1989] and domain vectors [Perrizo et al., 1991] or bitmap indices [O'Neil and Graefe, 1995]. We will shortly describe the first one. A more detailed summary of data-structure-assisted join can be found in section 3 of [Mishra and Eich, 1992].

A *join index* is a binary relation with two foreign key attributes[6]. A tuple in the join index describes a pair of tuples that participate in the join result by referring to the tuples in the input relation through the respective foreign key attribute value. Alternatively, foreign key values can be replaced through physical addresses or any other logical value – a *surrogate* – that uniquely identifies a tuple in an input relation. Figure 3.15 shows the (equi-)join algorithm based on a join index. Similarly to the preceding sections, figure 3.16 shows the search strategy employed by the algorithm. There are no mis-hits because the join is virtually precomputed.

Once created, a join index has to be maintained which might cause a considerable overhead: each time the input relations are updated the referential integrity has to be checked in order to keep the join index consistent. With a growing join selectivity the size of the join index approaches that of the cartesian product. This expense must be justified by frequent joins of the relations involved.

[6] or, more generally, two foreign key attribute sets, as a key can consist of more than one attribute.

```
/* There is a join index X for R and Q  */
/* and the join condition R.A = Q.B     */

for each tuple x in the join index X do
    get r ∈ R such that r.A = x.A
    get q ∈ Q such that q.B = x.B
    put r ∘ q in the output relation
od
```

Figure 3.15: Join algorithm based on a join index.

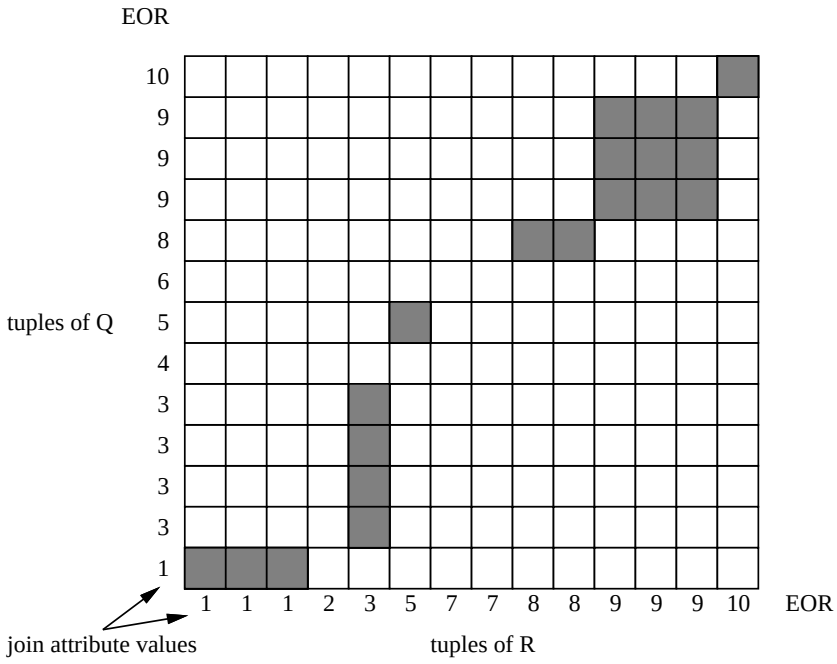


Figure 3.16: Search strategy of the join-index based algorithm.

3.5 Parallel Joins

Section 3.2.2 showed that the join operation is very performance-critical because of the large amounts of data that are involved. Section 3.4 then described several ways in which the join can be implemented efficiently on a sequential machine. An additional possibility is to parallelise the join. [Graefe, 1993] identified two alternative techniques:

- fragment-and-replicate,
- symmetric partitioning.

Many of the techniques found in commercial DBMS products fall into these two categories.

3.5.1 Fragment-And-Replicate Technique

The *fragment-and-replicate* (f-a-r) strategy partitions only one relation and replicates the other for joining it with each fragment:

$$R \bowtie_C Q \;=\; R_1 \bowtie_C Q \cup \cdots \cup R_m \bowtie_C Q \tag{3.5}$$

This method is particularly useful if R is huge and Q is small. This is a situation that occurs in what is frequently referred to as a *star-join* [Red Brick Systems, 1995b]. An advantage is that there are no constraints on the R_k. Any partition of R into subsets $R_1, \ldots, R_m$ will suit, especially a partition of R that might reside on the disks in the case of a parallel I/O system. This also means that this technique need not be affected by data skew. For this reason, load balancing is fairly easy to achieve.

The major drawback, however, is that Q is required to be small in order to keep replication costs marginal. If this is not the case then shipping Q over the interconnect of the parallel hardware can become the major bottleneck.

Depending on the join algorithm that is employed for processing the partial joins $R_k \bowtie_C Q$ we get various search patterns. Figures 3.17 and 3.18 show the ones that arise when using a nested-loop and a sort-merge join respectively.

3.5.2 Symmetric Partitioning Technique

A more general, but also more delicate parallel joining technique is based on *symmetric partitioning* where all participating relations are partitioned. We have

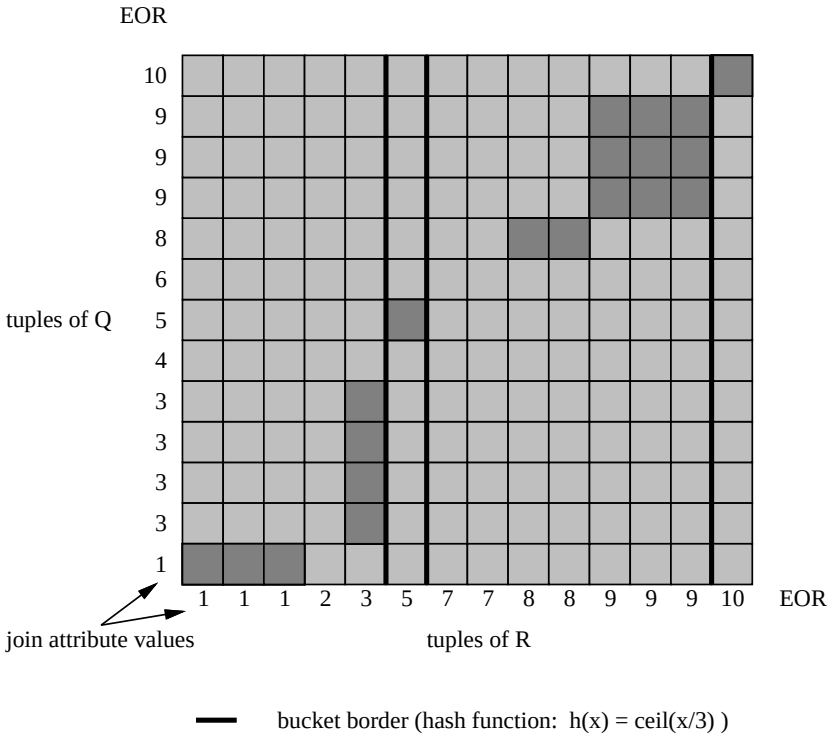


Figure 3.17: Search strategy of the fragment-and-replicate technique with the partial joins performed as nested-loops.

encountered this method already in the discussion of the Grace hash join (figure 3.14). Symmetric partitioning splits one 'big' join into several smaller *and* independent ones:

$$R \bowtie_C Q \;=\; R_1 \bowtie_C Q_1 \cup \cdots \cup R_m \bowtie_C Q_m \tag{3.6}$$

where the R_k and Q_k are referred to as *fragments* of the relations R and Q, respectively.

The *partial joins* $R_k \bowtie_C Q_k$ are independent from each other and can therefore be processed concurrently, i.e. in parallel. This parallelisation technique can be applied to *each* join algorithm that was presented in section 3.4. The Grace hash join in figure 3.14, for example, can be considered as a parallel hash join if the '**for** $k = 1$ **to** m **do**' loop is parallelised, i.e. if its body is executed concurrently.

Figure 3.19 shows the structure of this family of parallel joins. It is divided into three stages:

1. In a **partitioning stage** the two input relations R and Q are partitioned into fragments $R_1, \ldots, R_m$ and $Q_1, \ldots, Q_m$ respectively such that tuples

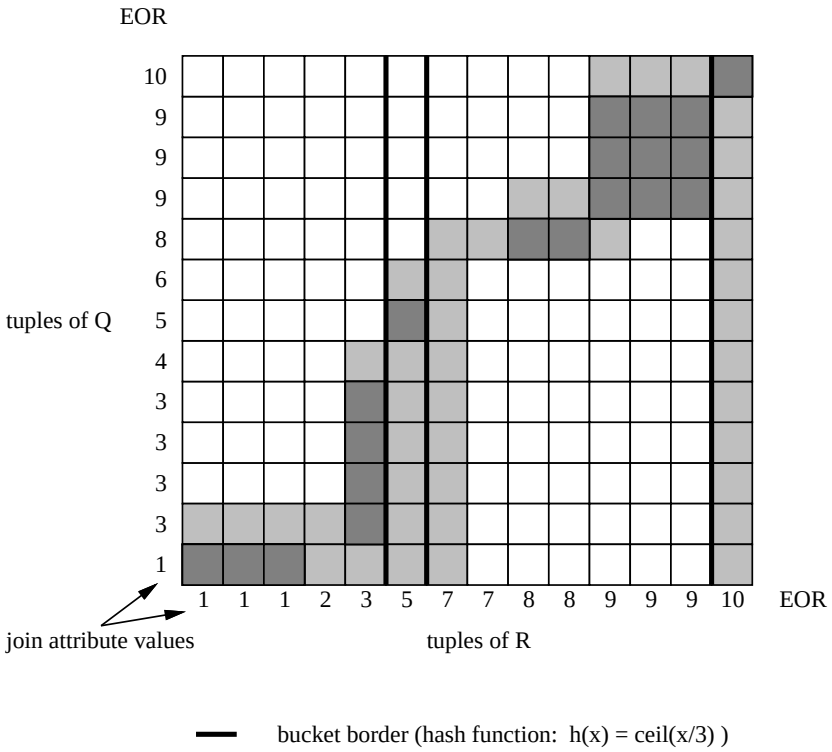


Figure 3.18: Search strategy of the fragment-and-replicate technique with the partial joins performed as sort-merge.

in any R_k can only join with tuples in Q_k. This secures the independence of the partial joins.

2. In a **joining stage** the partial joins $R_k \bowtie_C Q_k$ are executed in parallel (for $k = 1, \ldots, m$) which creates m local, partial results.

3. Finally, in a **merging stage** the partial results are merged (i.e. collected) to form the global join result.

Let us look at the search strategy of this family of parallel joins by using the same scenario as for the algorithms presented in section 3.4. The partitioning stage results in the same effect as encountered by the partitioning performed in the sequential hash joins. Therefore, figures 3.12 and 3.13 equally represent the search strategy of a parallel nested-loop join. Nevertheless, we add a further example in figure 3.20 because it makes certain issues more obvious: in this example, the join $R \bowtie_C Q$ is divided into four partial joins, each of which is processed by the nested-loop algorithm. This means that an exhaustive search is performed over four partial cartesian products $R_k \times Q_k$ for $k = 1, 2, 3, 4$. In terms of our graphic representation, this means that the four grey rectangular areas in figure 3.20 are processed.

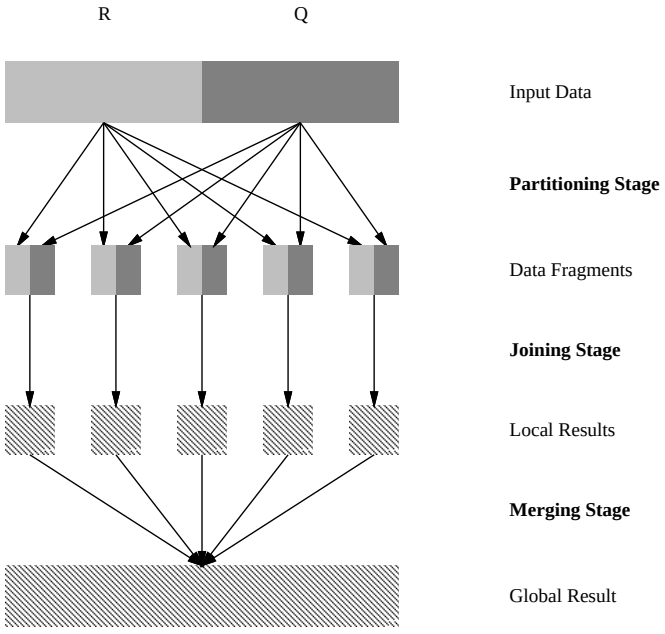


Figure 3.19: The structure of a parallel join based on symmetric partitioning.

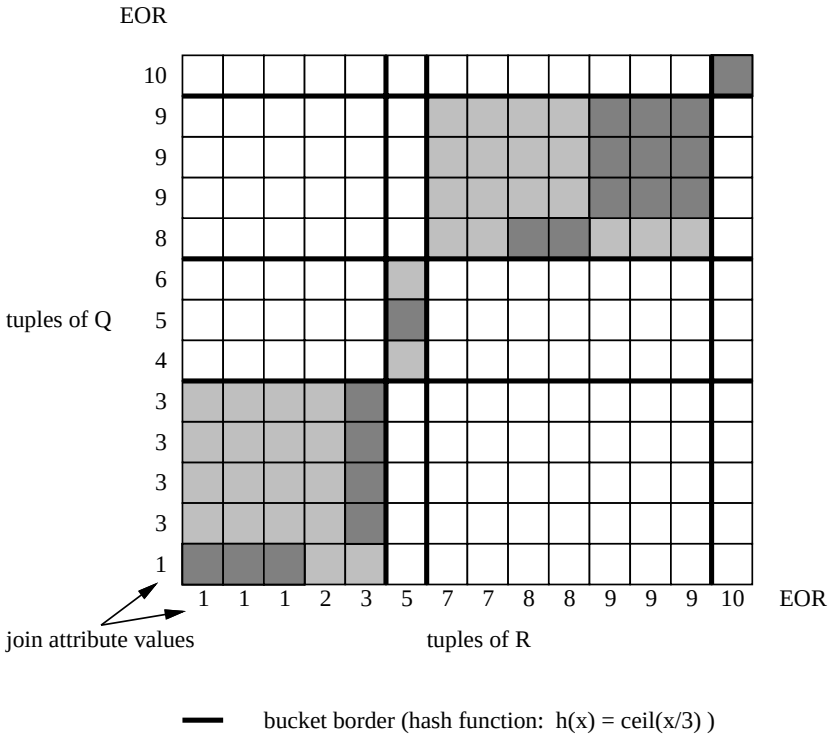


Figure 3.20: Search strategy of the symmetric partitioning technique with the partial joins performed as nested-loops.

The parallelisation, motivated by (3.6), works very well for equi-joins because it is easy to create *disjoint* fragments $R_1, \ldots, R_m$ ($Q_1, \ldots, Q_m$ respectively) which allow the partial joins to be independent. Unfortunately, many nonequi-joins and – as we will see in chapter 4 – many types of temporal joins cannot be divided into disjoint fragments *and* maintain the independence of the partial joins at the same time. One has then to decide whether to sacrifice either the disjointness or the independence with the first one being the preferable option in most cases.

The selectivities of the partial joins are much higher than for the original join because the partitioning concentrated tuples with similar values in associated fragments R_k and Q_k. This is an important point in the sense that the selectivity is an important issue for the choice of the most appropriate (sequential) join algorithm for computing the partial joins $R_k \bowtie_C Q_k$. In a set of experiments that we conducted on partitioned temporal joins, for example, we observed average selectivities of 80% for the partial joins of a join $R \bowtie_C Q$. Such high figures make the nested-loop algorithm the favourable option for processing the partial joins because there will not be a large number of unnecessary tuple comparisons and, at the same time, avoids any overhead caused

by sorting or hashing the data.

There are other ways of parallelising a join operation which differ from the ones presented in this section. The ones we presented are, however, the most common ones. One variation, for example, is to interleave the partitioning and joining stages similar to the case of sequential join algorithms. Further variations arise from different characteristics of the underlying parallel hardware architecture. Some parallel machines, for example, have specifically optimised communication facilities, such as broadcasts. This leads to specific cost models for this particular machine which might favour certain join strategies which are discarded when employing more general cost models. The interested reader might look at the papers that describe parallel join algorithms in detail, such as [Kitsuregawa et al., 1983], [DeWitt and Gerber, 1985], [Gerber, 1986], [Wang and Luk, 1988], [Schneider and DeWitt, 1989], [Kitsuregawa and Ogawa, 1990], [Wolf et al., 1990], [Keller and Roy, 1991] or [Wolf et al., 1993].

3.6 Classification of Join Algorithms

In order to classify the join algorithms that were presented in sections 3.4 and 3.5 we want to focus on the two main tasks that are performed by each join algorithm:

- the data is *partitioned* into fragments,
- the tuples of the fragments are *matched*.

The purpose of the partitioning stage is to reduce the number of pairs of tuples to be examined in the matching stage. The brute force nested-loop join of section 3.4.1 has no partitioning stage and therefore needs to test any possible tuple pair. This is the worst-case scenario. On the other hand, for example, there is the hash join which uses hashing as a way of partitioning the data before entering the matching stage. Similarly, the sort-merge join uses sorting as a way of partitioning. The type of partitioning employed is one important characteristic that distinguishes the join algorithms. Mishra and Eich identified four types of partitioning employed by join algorithms [Mishra and Eich, 1992]:

No Partitioning: The input relations are not partitioned at all. They must be exhaustively compared in order to find the tuple pairs that participate in the join.

Implicit Partitioning: Although the join algorithm does not have a specific step for performing the partitioning, it does do some dividing or ordering of the data in order to reduce the number of tuples to be compared in the match stage.

Explicit Partitioning: The algorithm contains an explicit partitioning stage as part of its execution.

Precomputed Partitioning: Partitioning is not performed as part of the actual join algorithm. These techniques assume that some partitioning exists.

In addition to the type of partitioning, another important characteristic is the mapping between the fragments of the input relations. Let us assume the case of an equi-join $R \bowtie_C Q$ with $C \equiv R.A = Q.B$. Suppose that R and Q are divided into their basic fragments $\dot{R}_1, \ldots, \dot{R}_m$ and $\dot{Q}_1, \ldots, \dot{Q}_m$ respectively with

m being the number of different values that actually occur in the attributes $R.A$ and $Q.B$. If we use the notation of relational algebra[7] this means that

$$m \;=\; |\pi_A(R) \cup \pi_B(Q)|$$

Imagine that the values in $R.A$ and $Q.B$ are $x_1, \ldots, x_m$. A *basic fragment* $\dot{R}_k$ then holds those tuples of R that hold x_k as the value in attribute A, i.e.

$$\dot{R}_k \;=\; \sigma_{A=x_k}(R)$$

The $\dot{Q}_k$ are defined accordingly. As an example for basic fragments you might look at figure 3.13 where complete partitioning created the basic fragments.

During the matching stage, each join algorithm overlaps a basic fragment, say $\dot{R}_k$, with one or more basic fragments of Q. The following degrees of overlap can be identified; please note that the definitions differ slightly from those presented in [Mishra and Eich, 1992] which is not clear enough in several aspects:

Complete Overlap: In this case, a basic fragment $\dot{R}_k$ meets all basic fragments of Q. This happens in the brute force nested-loops join (figure 3.6) or the nested-loop join in conjunction with the fragment-replicate-technique for parallelising a join (figure 3.17).

Minimum Overlap: Tuples of $\dot{R}_k$ meet all tuples of $\dot{Q}_k$ plus one tuple of $\dot{Q}_{k-1}$ and one of $\dot{Q}_{k+1}$. This kind of overlap is used in the sort-merge equi-join (figure 3.8).

No Overlap: Tuples of $\dot{R}_k$ only meet the tuples of $\dot{Q}_k$, e.g. in the completely partitioned hash join example of figure 3.13.

Disjoint Overlap: This occurs when the definition of the 'no-overlap-degree' is extended to *disjoint* fragments, with a fragment being the union of basic fragments. This means that tuples of a fragment, say $\dot{R}_{k_1} \cup \dot{R}_{k_2} \cup \cdots \cup \dot{R}_{k_i}$ with $\{k_1, k_2, \ldots, k_i\} \subset \{1, \ldots, m\}$, meet tuples of the corresponding fragment of Q, i.e. $\dot{Q}_{k_1} \cup \dot{Q}_{k_2} \cup \cdots \cup \dot{Q}_{k_i}$. This situation occurs in hash equi-joins (figures 3.12, 3.13 and 3.20).

Variable Overlap: Tuples of $\dot{R}_k$ meet tuples of $\dot{Q}_k$ and a varying number of 'neighbour fragments', i.e. $\dot{Q}_{l(k)}, \ldots, \dot{Q}_{k-1}, \dot{Q}_{k+1}, \ldots, \dot{Q}_{r(k)}$ with $1 \leq l(k) < k < r(k) \leq m$. We note that the values of $l(k)$ and $r(k)$ depend on k.

[7] See [Korth and Silberschatz, 1991] or [Lockemann et al., 1993] for example.

Algorithm	Type of Partitioning	Degree of Overlap
nested-loop	none	complete
sort-merge equi-join	implicit	minimum
sort-merge nonequi-joins	implicit	minimum / variable
hash equi-join	explicit	disjoint / no
join index	precomputed	no
parallel equi-join	explicit (symmetric)	disjoint / no
parallel equi-join	explicit (f-a-r) & no	complete
parallel equi-join	explicit (f-a-r) & implicit	variable

Table 3.1: Join algorithms, their type of partitioning and the degree of overlap.

Variable overlaps occur in many index-based join algorithms, some sort-merge nonequi-joins and sort-merge equi-joins in conjunction with the fragment-and-replicate strategy (see figure 3.18).

Table 3.1 summarises the relationships between join techniques, types of partitioning and degrees of overlap. There exists a certain duality between partitioning and overlap: no partitioning, for example, imposes a complete overlap in the matching stage. Explicit partitioning usually leads to a disjoint overlap and, in the extreme case, to no overlap. Figure 3.21 shows the categorisation of join algorithms that arises from that. We will use it in the following chapter when we discuss the implications given by temporal join conditions.

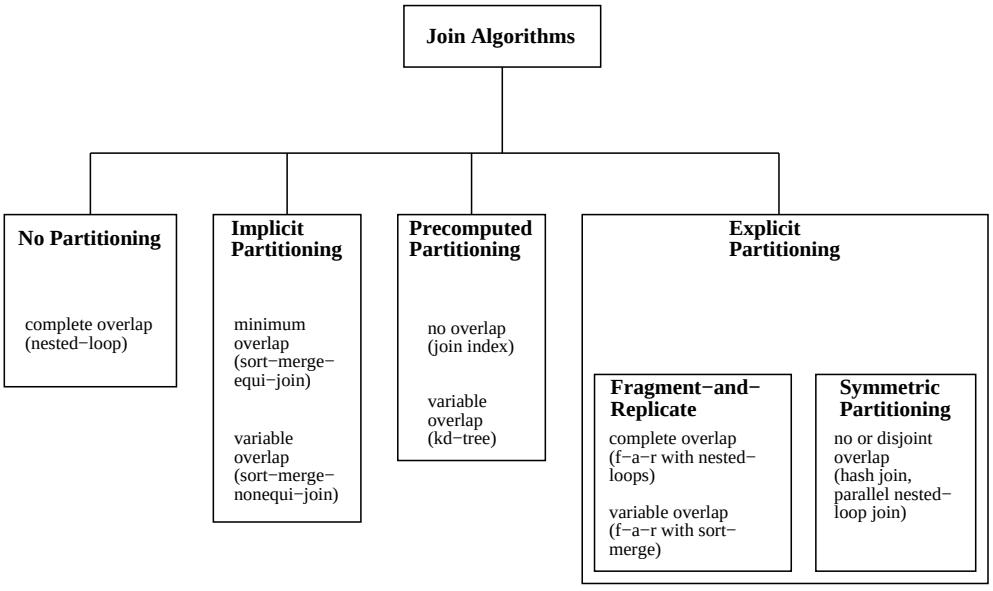


Figure 3.21: Join algorithm categorisation.

Chapter 4

Temporal Join Processing

In the previous chapter, join processing has been discussed in a very general context although there was an emphasis on equi-joins as the most frequently used join type. In this chapter, we want to focus on temporal joins.

Temporal joins have an impact on many of the aspects that were discussed in chapter 3. Consequently, many of the techniques that were designed and used for processing equi-joins are not directly applicable to temporal joins if a reasonable performance is required. In this chapter, we point to the differences, present adaptations for sequential temporal join processing that were presented in the literature, and propose improvements.

The issues discussed in this chapter are similar to those of chapter 3: Section 4.1 defines the temporal join operation and introduces a classification scheme for temporal join conditions. Here, the temporal intersection join is identified as a supertype of most other temporal joins. In section 4.2, we discuss the significance of the temporal join operation. Sections 4.3 and 4.4 present temporal join processing techniques. The discussion is divided into non-explicit partitioning (section 4.3) and explicit partitioning techniques (section 4.4). The first set of algorithms are straightforward adaptions of the corresponding equi-join techniques Here, modifications are only minor. Explicit partitioning algorithms, however, require certain parts of the relations to be replicated. This introduces an overhead in various ways. In section 4.4, we present techniques which reduce the overhead. Some of these have been proposed in the literature, some of them are new. In section 4.5, we conduct a simple comparison of temporal join algorithms. This allows us to summarise the most important features. Finally, in section 4.6, we focus on optimisation problems that are specific to temporal join processing.

4.1 Definition and Types of Temporal Joins

A join condition is said to be *temporal* if it enforces a certain relationship between the timestamps of the participating tuples. Hence, *temporal joins* combine two (or more) temporal relations using a temporal join condition. In terms of interval timestamps, this means that the intervals are related to each other. In chapter 2, we have already seen a typical example of such a temporal join condition, namely the intersection of two intervals as described by equation (2.3).

There are many possible relationships between two intervals: one interval can lie completely *before* the other, both intervals can *start* and / or *end* at the same time, they can *overlap* each other etc. Temporal joins can be classified according to the type of relationship on which its join condition is based. Table 4.1 shows a set of join conditions. We treat them as elementary for the following three reasons:

- they translate into simple expressions that describe the relationships between intervals' start- and endpoints,
- they, nevertheless, have reasonable semantics, that can be related to natural language expressions of everyday use,
- they can be used to compose more complex types, such as those in table 4.2.

An alternative set of elementary interval relationships was presented in [Allen, 1983] for the purpose of natural language processing. It leads, however, to more complex expressions between the start- and endpoints of the intervals that are involved. This makes it more difficult to decompose complex temporal join conditions into elementary ones. For that reason we prefer to use the set presented in table 4.1.

The literature has mainly concentrated on the intersection of intervals in the join predicate join. The reason behind this is that it requires the timestamps of the participating tuples to share at least one chronon. This is a minimum constraint that can be found in most other temporal join conditions (see tables 4.1 and 4.2).

Leung and Muntz referred to this minimum constraint as the 'TSJ1 query property' [Leung and Muntz, 1992]. Thus intersection join queries are identical with the TSJ1 queries discussed by Leung and Muntz. They show what optimisations (with respect to reducing tuple replication in the case of partitioning

Relationship	Join Name & Symbol	Condition	Informal Description
start	start join: $\overset{\text{sta}}{\bowtie}$	$r.t_s = q.t_s$	same timestamp start-points
finish	finish join: $\overset{\text{fin}}{\bowtie}$	$r.t_e = q.t_e$	same timestamp end-points
meet	meet join: $\overset{\text{mt}}{\bowtie}$	$r.t_e = q.t_s$	timestamp of r ends where timestamp of q starts, i.e. they meet.
before	before join: $\overset{\text{bef}}{\bowtie}$	$r.t_e < q.t_s$	timestamp of r comes before q's timestamp
left-overlap	left-overlap join: $\overset{\text{lo}}{\bowtie}$	$r.t_s > q.t_s \ \wedge \ r.t_s < q.t_e$	startpoint of r's timestamp lies within q's timestamp
right-overlap	right-overlap join: $\overset{\text{ro}}{\bowtie}$	$r.t_e > q.t_s \ \wedge \ r.t_e < q.t_e$	endpoint of r's timestamp lies within q's timestamp

Additional constraints are: $r.t_s \leq r.t_e \ \wedge \ q.t_s \leq q.t_e$

Table 4.1: Elementary temporal joins and respective conditions for joining tuples $r \in R$ with $q \in Q$.

over the interval timestamps) can be drawn from specialising the general intersection condition to, for example, a contain or during condition. In these cases, tuple replication can be restricted to some of the participating relations, e.g. to R in a contain join $R \overset{\text{con}}{\bowtie} Q$. However, replication is necessary for at least one of the participating relations in most temporal join conditions if the join is to be processed by partitioning over the interval timestamp attribute[1]. To summarise: whereas Leung and Muntz identify the situations in which tuple replication can be restricted, we will concentrate on how the impact of replication can be reduced *when replication is necessary*. The latter applies to a wide set of intersection based join conditions. Therefore we will focus on the temporal intersection join as the most general of these joins[2]. However, we stress

[1]Exceptions are those temporal join conditions that involve an equality relationship between interval start- and endpoints, such as the start or meet joins in table 4.1. These can be processed by one of the conventional equi-join techniques.

[2]Alternative names for the *temporal intersection join* are *T-join* [Segev, 1993] and *time-join* [Rana and Fotouhi, 1993]. Segev's *TE-join* includes, apart from the intersection condition, an equi-join condition, as does the *valid-time natural join* [Clifford and Croker, 1987], [Soo et al., 1994], the *natural time-join* [Clifford and Croker, 1987] and the *time-intersection equi-join* [Segev, 1993].

Join Name	Composition	Informal Description
equal join	$R \overset{=}{\bowtie} Q = R \overset{sta}{\bowtie} Q \cap R \overset{fin}{\bowtie} Q$	same timestamps
after join	$R \overset{aft}{\bowtie} Q = Q \overset{bef}{\bowtie} R$	timestamp of $r \in R$ is required to lie entirely after the timestamp of a $q \in Q$
overlap join	$R \overset{olp}{\bowtie} Q = R \overset{lo}{\bowtie} Q \cup R \overset{ro}{\bowtie} Q$	timestamps overlap but do not start or finish at the same point
contain join	$R \overset{con}{\bowtie} Q = (R \overset{lo}{\bowtie} Q \cap R \overset{ro}{\bowtie} Q) \cup (R \overset{sta}{\bowtie} Q \cap R \overset{ro}{\bowtie} Q) \cup (R \overset{lo}{\bowtie} Q \cap R \overset{fin}{\bowtie} Q)$	timestamp of an $r \in R$ contains the entire timestamp of a $q \in Q$
during join	$R \overset{dur}{\bowtie} Q = Q \overset{con}{\bowtie} R$	timestamp of an $r \in R$ is required to lie entirely within the timestamp of a $q \in Q$
intersection join	$R \overset{int}{\bowtie} Q = R \overset{lo}{\bowtie} Q \cup R \overset{ro}{\bowtie} Q \cup R \overset{sta}{\bowtie} Q \cup R \overset{fin}{\bowtie} Q \cup R \overset{mt}{\bowtie} Q \cup Q \overset{mt}{\bowtie} R$	timestamps intersect

Table 4.2: Examples of temporal join types that can be derived from the elementary ones.

again that replication is not only relevant for the pure intersection join but it is equally important for many other temporal joins, such as the during, contain, left-overlap, right-overlap and overlap joins. The results that are obtained for the case of the intersection join are therefore easily transferable to these joins.

Usually the tuples that satisfy the join condition are concatenated. In the case of temporal joins this concatenation is not trivial as the value of the timestamp for the resulting tuple has to be defined. This definition depends on the type of the temporal join; assuming temporal intersection the resulting timestamp is defined to be the intersection of the individual timestamps of the participating tuples. For example in the case of the two tuples r and q the resulting timestamp is

$$[\max\{r.t_s, q.t_s\}, \min\{r.t_e, q.t_e\}] \tag{4.1}$$

4.2 Significance of Temporal Joins

Temporal joins occur in queries that have to relate data from at least two relations in a temporal context. The most frequent temporal context is probably 'same time', such as facts that are required to be valid at the same time. Translated into relationships between interval timestamps this frequently means

- that the intervals have to be the same (*equality* condition), e.g. the period of a car rental and the period of the corresponding insurance policy,
- that one interval should lie within the other (*during* condition), e.g. the period of a car rental falling entirely in a low-season-pricing period,
- that the intervals have to share at least one chronon (*intersection* condition), e.g. the period of a car rental intersecting with a period of new fuel prices.

Trend analysis is an area in which simultaneity is an important aspect: if a trend – described by one set of data – is allegedly caused by a certain time-varying process – described by a second set of data – then this data can very often be related only by their date of occurrence or validity. In terms of the relational data model this means nothing but performing a temporal intersection join on the two data sets.

Imagine, for example, a manager who analyses the summer sales figures and tries to find out why sales went up this year. He might want to relate the sales with the hotter weather, the general rise in wages, the favourable exchange rate, the floatation of several building societies (and its implication that there is more money in consumers' pockets) or other facts that can influence customer behaviour. Assume that such data is kept in various relations. One of the few possibilities to associate the data from the various sources is to relate them temporally, using a 'same time' context. In terms of query processing this translates into executing temporal joins.

In fact, many analysis attempts that seek to confirm causality between two or more trends or effects have to start by relating the underlying data in terms of their temporal context. This only reflects the way in which we analyse many things ourselves: whenever we investigate the cause of a certain effect we try to figure out what other effects / events happened at the *same time*[3], possibly

[3] Actually this is only a special situation for the case that the cause is immediately followed by the effect. There are many examples for which the effect is delayed for a certain constant period. Such constant delays are not a problem and can be incorporated in the join condition without changing any of the issues that have been elaborated for the '*same time*' notion.

at the same place.

For that reason, relating data temporally is a feature which is supported by many decision support tools. Such tools are usually part of a decision support systems (DSS) which itself is frequently built on top of a data warehouse (see figure 2.5). In other words: decision support tools are liable to create complex queries involving temporal joins. These queries have to be efficiently supported by the underlying database technology. This is one aspect of the relationship between temporal databases and data warehouses which we already elaborated in section 2.4.

4.3 Non-Explicit-Partitioning Techniques

4.3.1 Overview

In this section, many of the issues and results that have been discussed in the previous chapters and sections are brought together. We will look at temporal join algorithms that are *not* based on explicit partitioning, i.e. algorithms that do not contain an explicit partitioning stage as an integral part. This group of algorithms comprises nested-loop joins (with no partitioning stage), sort-merge joins (with an implicit partitioning stage) and index joins (using pre-computed partitioning). Figure 4.1 illustrates these three techniques within the hierarchy of section 3.6. In chapter 3, we discussed these techniques in the light of processing equi-joins. In this section, we look at them for processing temporal joins.

As we can expect from the issues presented in section 3.4, it is only the sort-merge approach that needs some minor modifications. The general idea, however, remains unchanged. Nested-loops and join indices do not require any change. Nevertheless, temporal join conditions imply increased join selection ratios in general. This means that join results are larger in comparison to equi-joins between equally sized relations. Or put differently: a bigger share of the cartesian product participates in the temporal join result. This, however, might influence the choice of algorithm as we have already indicated in chapter 3.

In the following discussions, we will concentrate on the temporal intersection join whenever the type of the temporal join is relevant. As we mentioned in section 4.1, the intersection join can be considered as a supertype of most temporal joins. Algorithms for the more specialised temporal joins can be derived from the intersection join algorithms.

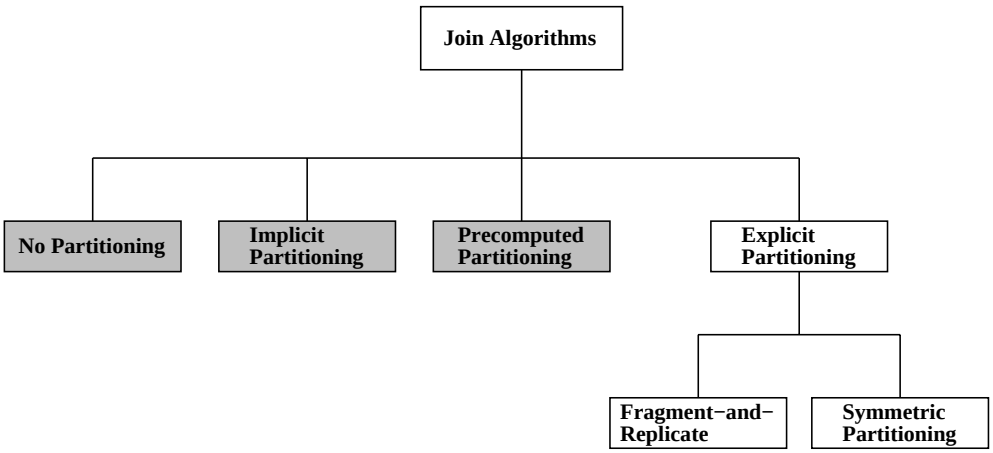


Figure 4.1: Non-explicit partitioning joins.

4.3.2 Nested-Loop Temporal Joins

The basic nested-loop join, as outlined in figure 3.6, does not need to be adapted for processing temporal join conditions. It checks every tuple pair of the cartesian product anyway. Figure 4.2 shows an example of a temporal intersection join between two relations which are equal in size to the ones used in the example in chapter 3. The figure indicates that more tuple comparisons are successful than in the case of a nested-loop equi-join with equally sized relations. The nature of temporal join condition means that temporal joins (and especially temporal intersection joins) may frequently produce much higher join selectivities than comparable equi-joins. This means that an exhaustive search performed by a nested-loop join algorithm is possibly not as adverse as in the case of an equi-join.

The selectivities resulting from the temporal join condition considered here are more likely to lead to situations in which even small inputs can produce huge results. Such huge results are impractical to handle in both cases, as an end result but also as an intermediate result. An optimiser will therefore try to avoid to compute such gigantic joins either by warning the user about a possible huge end result (the user can then change or dismiss the query) or by rearranging the sequence in which the query operations are processed (such that a huge intermediate join result can be by-passed).

It is almost impossible to say what typical temporal join selectivities are. This is (a) due to the variety of temporal join conditions (see tables 4.1 and 4.2, for example) and (b) due to the great variety in the statistical characteristics

of temporal applications[4] which makes it already impossible to determine a typical selectivity for a pure intersection condition. The latter, for example, would depend on the typical interval length (which can vary widely between different applications), whether interval startpoints are spread uniformly over the lifespan (e.g. phone calls over a day) or whether they come in clusters (e.g. hourly measurements of certain parameters) and also the granularity of the time dimension.

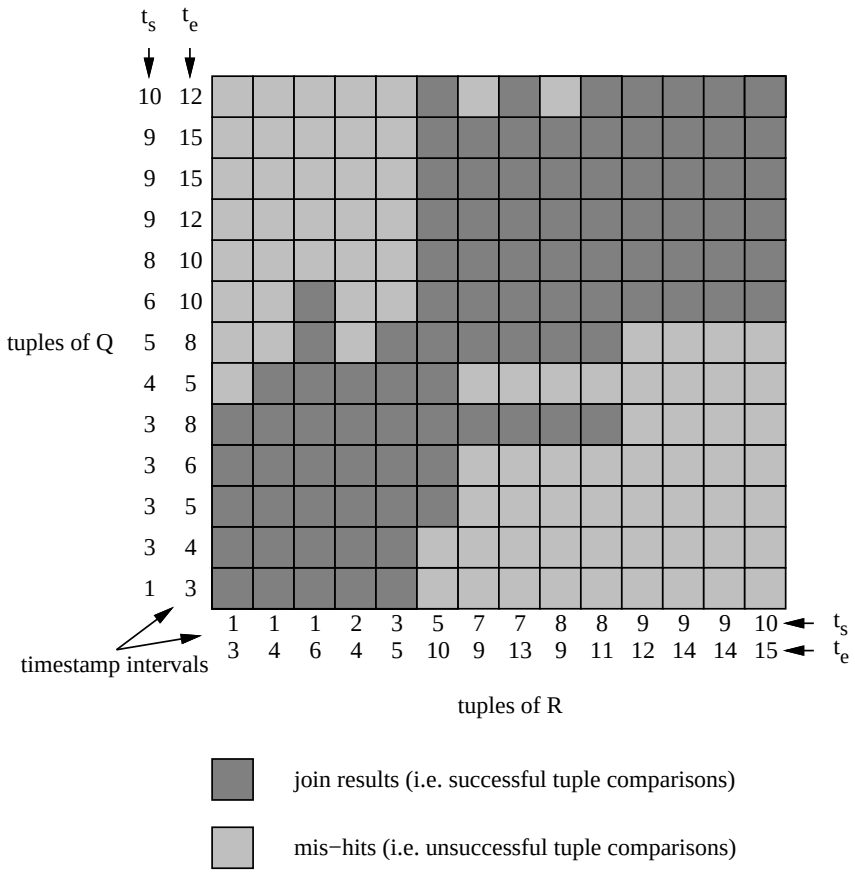


Figure 4.2: Search strategy for a nested-loop temporal intersection join.

4.3.3 Sort-Merge Joins

The actual shape of the sort-merge join algorithm depends on the underlying join condition. In section 3.4.2, we outline the sort-merge join algorithm for an equi-join condition. For a temporal join condition, such as temporal intersection, the general structure remains the same: a sorting stage is followed by a

[4]See discussion in section 10.1 on this issue.

merging stage. The merging stage, however, needs to be modified.

Most of the algorithms that have been proposed for temporal join processing employ a sort-merge strategy. Examples can be found in [Gunadhi and Segev, 1990], [Leung and Muntz, 1990], [Gunadhi and Segev, 1991], [Rana and Fotouhi, 1993] and [Segev, 1993]. Many authors consider their algorithms as refinements of the nested-loop approach that take advantage of the fact that one or all relations are sorted in ascending or descending order. This means that they merely discuss the merging stage of the sort-merge approach and assume that the required sort order is either enforced by a preceding sorting stage or already exists. The merging stage, however, can be regarded as nested loops in which the inner loop takes advantage of information that was obtained during previous loops.

Assuming the relations to be sorted can be reasonable, especially in the case of transaction time relations. Here, tuple timestamps are created according to the time of the update (i.e. the transaction time). If these tuples are appended to the end of the relation we get a natural sort order of the tuples. This is sometimes called the *append-only characteristic* of transaction time relations.

Figure 4.3 shows a sort-merge algorithm for a temporal intersection join. Originally, it was discussed as *Algorithm 2* in [Rana and Fotouhi, 1993] which is similar to algorithm *TJ-1* in [Gunadhi and Segev, 1991]. The two **if**-statements in the inner loop check the two situations in which no intersection occurs:

- Situation 1: r's timestamp lies *before* q's timestamp, i.e. $r.t_e < q.t_s$, and
- Situation 2: r's timestamp lies *after* q's timestamp, i.e. $r.t_s > q.t_e$.

In the first situation, the inner loop (which scans relation Q) has to be left in order to proceed with R in the outer loop. In the second situation, we have to proceed with Q. If all previous checks in the inner loop have been unsuccessful, i.e. if *flag* $=$ false, then the 'start tuple marker' q_{start} can be increased, too. If none of the two situation occurs then the timestamps intersect and the concatenation of the two tuples can be placed in the result. The specific characteristic of the concatenation are described by (4.1).

Figure 4.4 shows the search strategy of this algorithm. The outer loop moves along the horizontal axis whereas the inner loop scans vertically, along the column designated by the outer loop. Information obtained from previous loop runs provide a hint as to the best entry point in the column, i.e. several tuples at the beginning (coming from the bottom) might be omitted.

```
/* Stage 1: Sorting */
sort R on R.t_s as primary key and R.t_e as secondary key
sort Q on Q.t_s as primary key and Q.t_e as secondary key

/* Stage 2: Merging */
r       = first tuple in R
q_start = first tuple in Q

while r ≠ EOR and q_start ≠ EOR do
    q    = q_start
    flag = false

    while q ≠ EOR do
        if r.t_e < q.t_s then
            leave inner loop
        fi
        if r.t_s > q.t_e then
            q = next tuple in Q after q
            if not flag then
                q_start = q
            fi
        else
            put r ∘ q in the output relation
            flag = true
            q    = next tuple in Q after q
        fi
    od

    r = next tuple in R after r
od
```

Remark: *EOR* denotes an 'end-of-relation' mark that is returned by either the 'first tuple in' or the 'next tuple in' operation if it fails to retrieve a tuple because the end of the respective relation has been reached.

Figure 4.3: Sort-merge temporal intersection join algorithm.

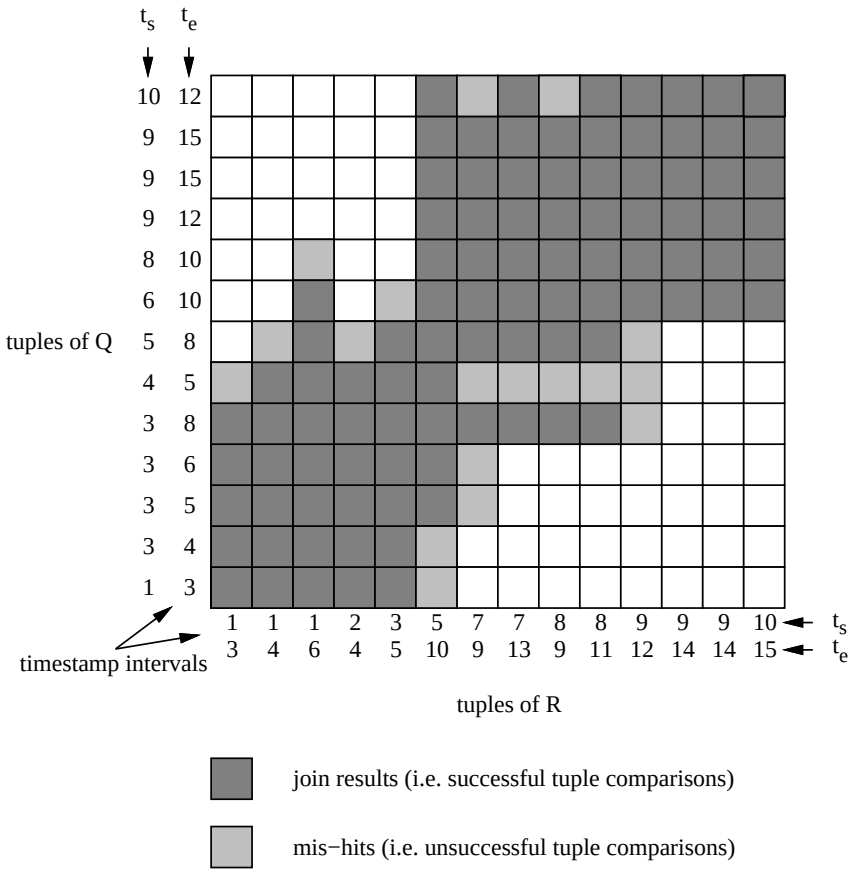


Figure 4.4: Search strategy of the sort-merge temporal intersection join of figure 4.3.

4.3.4 Data-Structure-Assisted Joins

The join techniques that are assisted by data structures, such as index trees or bitmap indices, might require modification in order to process interval data. B-trees [Bayer, 1972], for example, are designed for indexing atomic data on which a total order is defined. The total order is essential for this indexing method and most other conventional ones, too. However, there is no total ordering among interval data which would enable us to find separating values x such that an interval is either less than or equal to x or greater than x. Unfortunately, one can always find a third type of interval which just overlaps the breakpoint x and is neither less-equal nor greater than x. This makes it impossible in the general case to find a partition (of the time line) that satisfies certain optimality constraints. A lot of index methods, however, rely on such optimal partitions. Consider, for example, the process of balancing a B-tree which effectively means adjusting the original partition (of the indexing attribute's domain) in order to achieve a better balance of the tree.

Elmasri *et al.* proposed an indexing method, called the *time index*, which can be used for join processing purposes [Elmasri et al., 1993]. In general, various indexing methods have been proposed for interval timestamps. Further examples can be found in [Kolovson, 1993] and [Gunadhi and Segev, 1993]. They all can be used with the general join algorithm in figure 4.5.

Similarly, the join index based algorithm presented in figure 3.15 works unmodified given that the underlying join index is built upon the temporal intersection condition.

In both cases, the search strategy should be the same and avoid any unsuccessful tuples comparisons and unnecessary retrievals. Figure 4.6 visualises the search strategy for the example that has been employed throughout this chapter.

```
/* there is an index I of Q over the timestamp attribute */

for each tuple r in R do

    Use I to identify those q ∈ Q whose timestamp intersects
    with the one of r
    put the references q⃗ to these q in set X

    for each q⃗ in X do
        q = tuple in Q which is referred to by q⃗
        put r ∘ q into the join result.
    od

od
```

Figure 4.5: Join algorithm using an index.

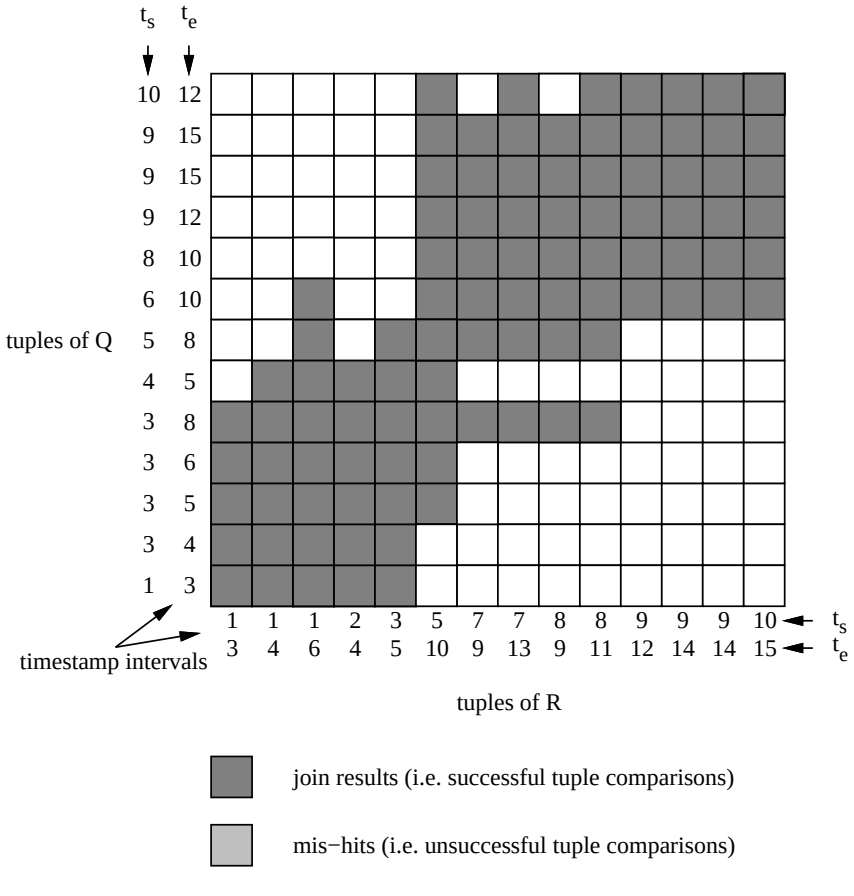


Figure 4.6: Search strategy of an index based temporal intersection join.

4.4 Explicit-Partitioning Join Algorithms

4.4.1 Overview

In this section, we consider join techniques that include an explicit partitioning stage as an integral part of the join algorithm. They are shown in grey boxes in figure 4.7. In the case of the equi-join these techniques have proved to be very efficient and very versatile, especially allowing the parallelisation of the join operation. We might expect them to be similarly successful in the case of temporal joins.

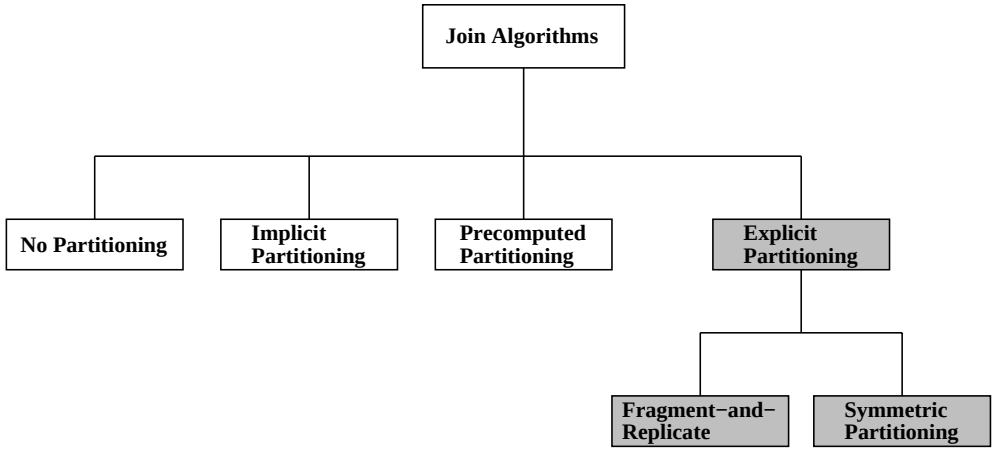


Figure 4.7: Explicit partitioning joins.

We will merely concentrate on the symmetric partitioning approach, i.e. both participating relations are partitioned; the fragment-and-replicate strategy that was presented in section 3.5.1 is not affected by the actual join condition and therefore works in the same way for temporal joins as it does for any other type of join.

The basic temporal join strategy employing symmetric partitioning is built upon equation (3.6). It will be discussed in section 4.4.2. This will reveal the problem that the fragments R_k ($k = 1, \ldots, m$) cannot be disjoint because of the temporal intersection condition. This leads to certain negative implications:

(a) **Replication Overhead:** During the partitioning process, a lot of tuples have to be (logically) replicated to be placed into several fragments. Please note that this logical replication does not necessarily translate into a physical replication: when working on a shared-memory machine tuple fragments might be represented as an index, i.e. a set of pointers that

refer to the actual locations in memory or on disk where the tuples are stored. In this case, the logical replication causes an additional effort when building these indices. When working on a shared-nothing architecture, however, logical replication is likely to be translated into a physical replication. In both cases, logical replication causes an overhead.

(b) **Processing Overhead:** Because of tuple replication, the individual fragments become larger. Hence, processing the partial joins $R_k \bowtie_C Q_k$ requires more effort. This causes a processing overhead.

(c) **Duplicates Overhead:** Tuple replication can also produce duplicates in the result which either cause a further overhead in subsequent stages of a query evaluation or which have to be removed which itself is a potentially expensive process.

We will present algorithms that partially tackle these effects:

- In section 4.4.2, a straightforward temporal adaption of the simple hash join is given. It suffers from all three overheads.
- In section 4.4.3, a join strategy is derived that reduces the processing overhead (b) and avoids the duplicates overhead (c). Effect (a), however, cannot be avoided when the partial joins have to be kept independent from each other for processing them in parallel. This strategy was originally proposed in [Zurek, 1996].
- In section 4.4.4, we present a strategy that was originally used in [Soo et al., 1994]. By sequentially processing the partial joins and keeping certain tuples in memory between each partial join evaluation one can avoid the replication overhead (a). Unfortunately, this method sacrifices the independence of the partial joins which therefore cannot be processed concurrently anymore.
- In section 4.4.5, a rather different approach is presented which is based on spatial partitions. It was proposed in [Lu et al., 1994] and maps intervals to points in a two dimensional space. This space is divided into disjoint parts which results in disjoint relation fragments. In this way, the replication and duplicates overheads, (a) and (c), are avoided. However, join processing requires a variable overlap of the fragments which either restricts the concurrency of the partial joins or requires fragments to be replicated, which means that the processing overhead (b) remains and a

replication overhead might have to be re-introduced for parallelisation purposes.

As mentioned above, we will concentrate on the temporal intersection join. Many of its subtypes allow optimisations such as restricting the replication of tuples to one relation. During or contain joins are examples for that. Leung and Muntz defined an *assymmetry property* in order to identify join conditions which lead into such situations [Leung and Muntz, 1992]. Here, however, we assume the situation of the intersection join in which both (or, more generally, *all*) participating relations require replication.

4.4.2 Simple Temporal Hash Join

The symmetric partitioning approach, as outlined by equation (3.6)

$$R \bowtie_C Q \;=\; R_1 \bowtie_C Q_1 \cup \cdots \cup R_m \bowtie_C Q_m$$

can be adapted for processing temporal intersection joins. We assume that the partial joins are to be kept independent thus allowing concurrent processing. Then, the fragments cannot be created by hashing the tuples over their interval timestamps: assume that there was a hash function h such that intersecting intervals are assigned to the same fragment number. Assume also that there are at least two different fragments per relation to be created. Let $[x_s, x_e]$ and $[y_s, y_e]$ be two non-intersecting intervals, i.e. $x_s \leq x_e < y_s \leq y_e$, which are assigned to two different fragments i and j by h, i.e.

$$h(x_s, x_e) = i \quad \neq \quad j = h(y_s, y_e)$$

Now consider the interval $[x_e, y_s]$ which should fall into fragment i because it intersects with $[x_s, x_e]$ and also into fragment j because it intersects with $[y_s, y_e]$. Thus h would have to assign two different values, i and j, for $[x_e, y_s]$ which contradicts its notion of being a function.

Nevertheless, we can employ *range partitioning* as a variation of hash partitioning. Here, tuples are partitioned over their interval timestamp in the following way: the time line is divided into m disjoint ranges which are numbered according to their order on the time line. If a relation, say R, is partitioned then a tuple $r \in R$ is put into R_k if its timestamp interval intersects with the k-th range. We note that a tuple can be put into several fragments because its timestamp interval might intersect with more than one time range.

In general, it is impossible to create disjoint fragments for interval data if a temporal relation R has a long 'chain' of intervals, i.e. $[r_1.t_s, r_1.t_e]$, ..., $[r_n.t_s, r_n.t_e]$ such that subsequent intervals intersect, i.e.

$$[r_i.t_s, r_i.t_e] \cap [r_{i+1}.t_s, r_{i+1}.t_e] \neq \emptyset$$

Then it is impossible to put these intervals into different fragments without assigning at least one of them to two different fragments: if $[r_1.t_s, r_1.t_e]$ goes to fragment k then $[r_2.t_s, r_2.t_e]$ has to go to fragment k, too, because it intersects with $[r_1.t_s, r_1.t_e]$. The same applies to $[r_3.t_s, r_3.t_e]$ because it intersects with $[r_2.t_s, r_2.t_e]$, and so forth. However, our intention is to partition the data in order to reduce the processing effort and to improve resource utilisation (disk I/O, main memory). It is therefore out of the question to put all the n intervals into the same fragment if n is very large; in general, it is likely that n is close or equal to $|R|$. In this case, the 'chain' has to be broken into two or more parts which forces one or more intervals to be put into more than one fragment.

If tuple replication is tolerated as a necessary evil, one can apply the symmetric partitioning technique as described in section 3.5.2. Figure 4.8 shows the search strategy of the technique when the partial joins are processed as nested-loops. It also shows two major problems related to the processing overhead and the duplicates overhead:

- Tuple replication causes repetitions of possible tuple comparisons. The net effect of this is the same as if the search space is increased. The fragments themselves are larger. In figure 4.8, 132 tuple comparisons are performed. When comparing this number with the 182 of the nested-loop join (figure 4.2) we recognise that the partitioning effort is not as successful as one might have hoped.

- Some of the successful tuple comparisons are duplicated too. These cause duplicate tuples in the join result. In figure 4.8, these comparisons are marked in black. They occur, say, in the k-th partial join $R_k \bowtie_C Q_k$, and are actually unnecessary because they are also performed by another partial join $R_i \bowtie_C Q_i$ with $i < k$. In many situations, duplicates have to be removed from the join result by expensive operations. This increases the overall overhead furthermore.

In the following section, we will tackle these problems by refining the simple approach presented here.

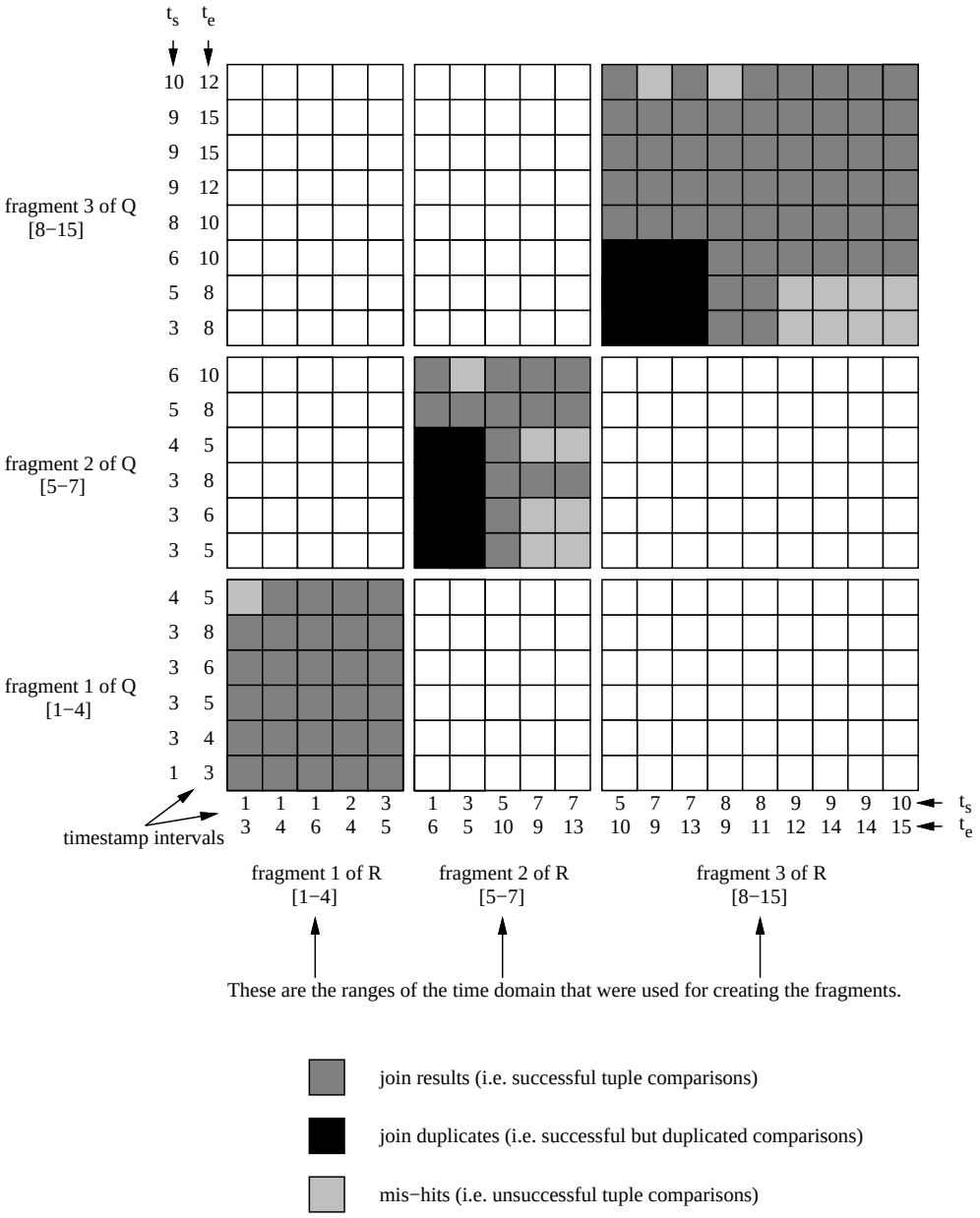


Figure 4.8: Search strategy for the simple partitioned temporal join (partial joins as nested-loops).

4.4.3 Improved Temporal Hash Join

Above, we identified successful but replicated tuple comparisons as a major problem of the simple approach. A first measure could be to extend the tuple comparisons by checking if the respective comparison occurs somewhere else. This avoids duplicates in the result but does not avoid the actual unnecessary tuple comparisons. Thus, it still produces a processing overhead. Consequently, replicated tuple comparisons should be avoided all together.

For that purpose, we observe the following: consider two tuples $r \in R$ and $q \in Q$ with respective timestamps $[r.t_s, r.t_e]$ and $[q.t_s, q.t_e]$. Assume that the timestamps intersect and therefore that r and q are compared with each other at some stage of the computation. If $r.t_s$ falls in the i-th and $q.t_s$ in the j-th range of the time domain then they are compared in the partial join $R_{\max\{i,j\}} \bowtie_C Q_{\max\{i,j\}}$ for the first time. Possibly, they are compared again in subsequent partial joins.

In order to avoid this, a fragment, say R_k, can be divided into two disjoint subfragments, R'_k and R''_k. R'_k holds the tuples whose timestamp startpoint falls into the k-th range – these are called the *primary tuples* – and R''_k holds tuples that are already in some fragment R_j with $j < k$ – these are called the *replicated tuples*, i.e.

$$\begin{aligned} R'_k &= \{r \in R_k : r.t_s \text{ falls into the } k\text{-th time range}\} \\ R''_k &= \{r \in R_k : r.t_s \text{ does not fall into the } k\text{-th time range}\} \end{aligned}$$

A partial join then looks like this

$$\begin{aligned} R_k \bowtie_C Q_k &= (R'_k \cup R''_k) \bowtie_C (Q'_k \cup Q''_k) \\ &= R'_k \bowtie_C Q'_k \cup R'_k \bowtie_C Q''_k \cup R''_k \bowtie_C Q'_k \cup R''_k \bowtie_C Q''_k \end{aligned} \quad (4.2)$$

However, the join $R''_k \bowtie_C Q''_k$ comprises exactly those unnecessary tuple comparisons that we have identified above; figure 4.9 illustrates this for the partial join $R_3 \bowtie_C Q_3$ of the example of figure 4.8. Processing can therefore be restricted to the first three joins in (4.2). We note that this restriction applies only when the entire join $R \bowtie_C Q$ is computed; for getting the result of $R_k \bowtie_C Q_k$ we still require $R''_k \bowtie_C Q''_k$ because

$$R'_k \bowtie_C Q'_k \cup R'_k \bowtie_C Q''_k \cup R''_k \bowtie_C Q'_k \subsetneq R_k \bowtie_C Q_k$$

if $R''_k \neq \emptyset$ and $Q''_k \neq \emptyset$. We note that $R''_k \overset{\text{int}}{\bowtie} Q''_k = R''_k \bowtie_{\text{true}} Q''_k$ due to the definition of R''_k and Q''_k. In total, $R \bowtie_C Q$ can be decomposed as shown in figure 4.10. This leads to the search strategy shown in figure 4.11.

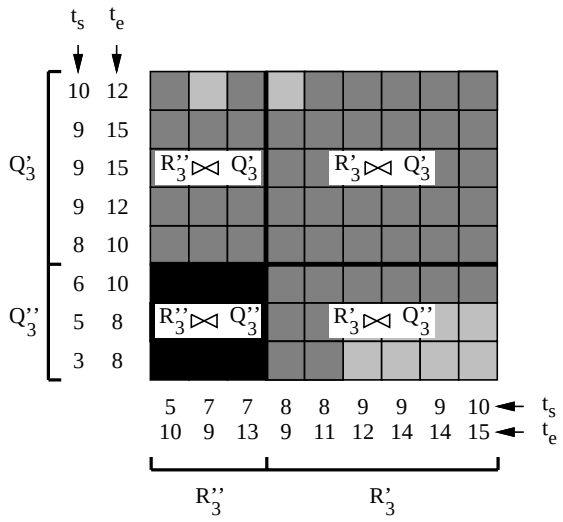


Figure 4.9: Illustration of equation (4.2) for $R_3 \bowtie_C Q_3$ in figure 4.8.

$$\boxed{R \bowtie_C Q}$$

$$\downarrow$$

$$\boxed{R_1 \bowtie_C Q_1 \cup \cdots \cup R_k \bowtie_C Q_k \cup \cdots \cup R_m \bowtie_C Q_m}$$

$$\downarrow$$

$$\boxed{\begin{array}{c} R'_1 \bowtie_C Q'_1 \cup R'_1 \bowtie_C Q''_1 \cup R''_1 \bowtie_C Q'_1 \\ \cup \cdots \cup \\ R'_k \bowtie_C Q'_k \cup R'_k \bowtie_C Q''_k \cup R''_k \bowtie_C Q'_k \\ \cup \cdots \cup \\ R'_m \bowtie_C Q'_m \cup R'_m \bowtie_C Q''_m \cup R''_m \bowtie_C Q'_m \end{array}}$$

Figure 4.10: Improved partitioning for computing a temporal intersection join.

A second optimisation is based on the following observation: if the remaining three joins in (4.2) are processed sequentially in one of the orders

1. $R'_k \bowtie Q''_k$, $R'_k \bowtie Q'_k$, $R''_k \bowtie Q'_k$ or
2. $R''_k \bowtie Q'_k$, $R'_k \bowtie Q'_k$, $R'_k \bowtie Q''_k$

and R'_k and Q'_k fit, respectively, into the local main memory of the processing node, then we can avoid unnecessary accesses to secondary storage. We can, however, enforce this situation because the sizes of the R'_k and Q'_k are much easier to predict and to control than those of the R''_k and Q''_k (and consequently those of R_k and Q_k): each tuple of R and Q appears exactly in one R'_k and Q'_k but might appear in several R''_k or Q''_k, respectively. The assignment of a tuple to a certain R'_k (Q'_k resp.) only depends on the value of the startpoint of the timestamp. This, however, is nothing else than partitioning atomic values as in the case of an equi-join. In the simplest case, the number of fragments m can be chosen high enough to fit the R'_k and Q'_k into main memory. More sophisticated techniques might be necessary if the startpoint values are heavily skewed, see e.g. [Hua and Lee, 1991] or [DeWitt et al., 1992].

4.4.4 Partitioned Temporal Join for Sequential Processing

The separation of primary and replicated tuples within an R_k can be exploited for sequential processing too. Soo *et al.* proposed the following strategy in [Soo et al., 1994]:

- The partial joins $R_1 \bowtie_C Q_1, \ldots, R_m \bowtie_C Q_m$ are processed either from 'left to right' (i.e. 1 to m) or 'right to left' (i.e. m to 1).
- Assuming that the partial joins are processed in the order from 1 to m, we can avoid the replication of tuples which, for example, takes place within a partitioning stage that precedes that joining stage. This can be done in the following way: after a partial join $R_k \bowtie_C Q_k$ has been processed the tuples $R''_{k+1} \subseteq R_k$ and $Q''_{k+1} \subseteq Q_k$ are kept in a cache memory. Thus, only R'_{k+1} and Q'_{k+1} need to be accessed on disk.

Figure 4.12 summarises this strategy. Again, if the partial joins are evaluated by nested loops we get a search strategy as in figure 4.8. Please note that the difference lies in the fact that, here, partial joins are not independent from each other and require sequential processing in order to save disk access costs. Soo *et al.*'s technique can also be improved by decomposing the partial joins as

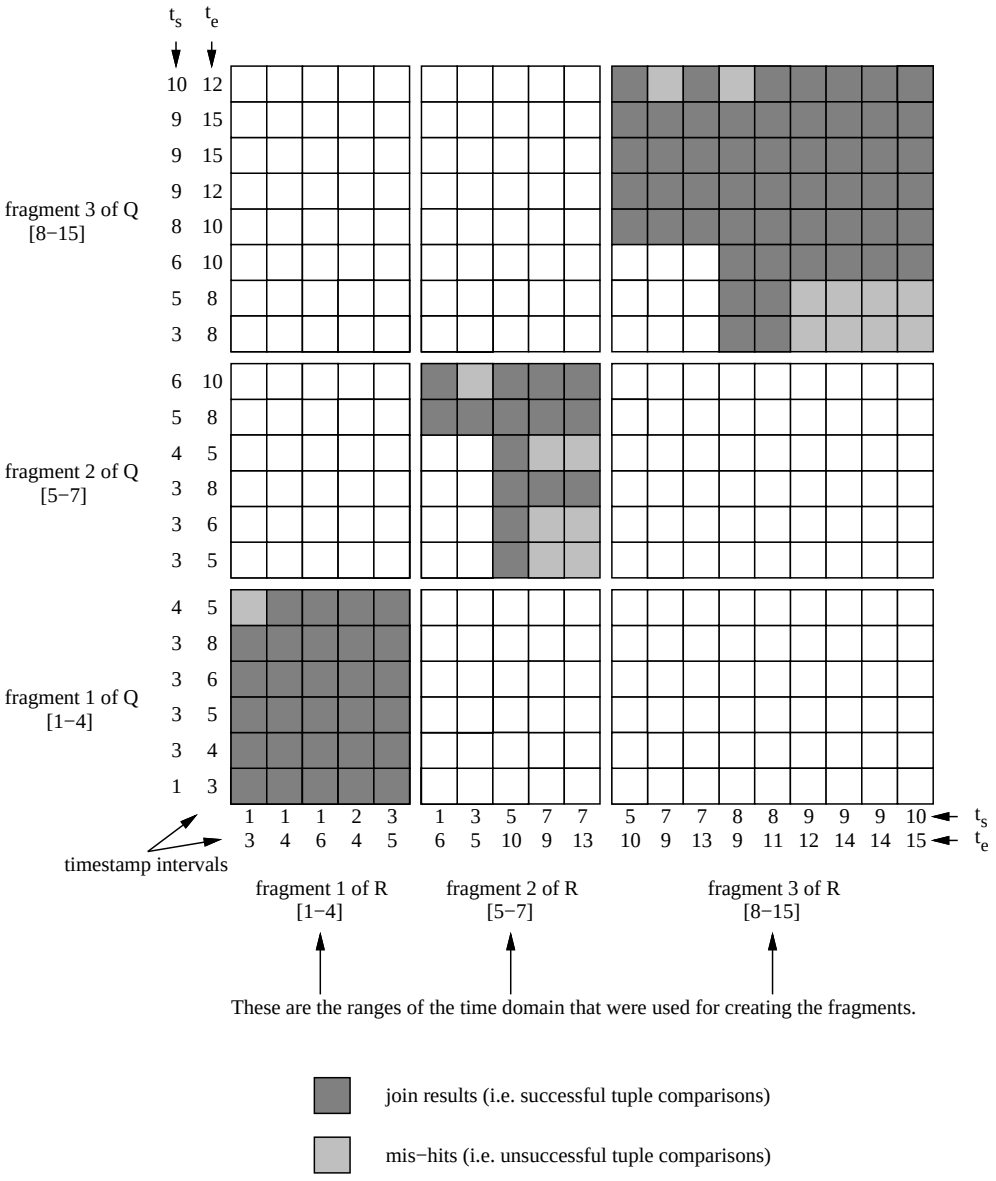


Figure 4.11: Search strategy for the improved partitioned temporal join.

described in section 4.4.3. The resulting search strategy then corresponds to figure 4.11.

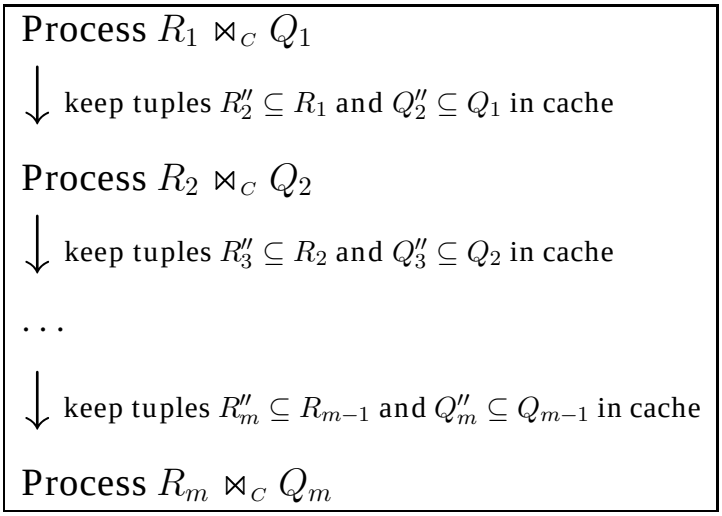


Figure 4.12: Sequential processing of a partitioned temporal intersection join.

4.4.5 Spatially Partitioned Temporal Join

In this section, we describe a partition scheme which was proposed by [Lu et al., 1994]. It creates *disjoint* fragments for both relations. Hence, it is a symmetric partitioning approach. On the other side however, a fragment of one relation needs to be joined not only with one fragment of the other but – depending on its index number – with a set of fragments. Thus, if this method is to be parallelised, fragments need to be replicated in order to achieve independent partial joins. From this point of view, one could also speak of a fragment-and-replicate approach.

Lu *et al.* use a spatial rendition which was described in [Hinrichs and Nievergelt, 1983]. An interval $[t_s, t_e]$ is mapped to a point in a two-dimensional grid. Its coordinates, x and y, are calculated by the equations:

$$\begin{aligned} x &= t_s \\ y &= t_e - t_s \end{aligned}$$

This means

- that an interval starting at time t has its corresponding grid point on the line $x = t$,

- that an interval ending at time t has its corresponding grid point on the line $x + y = t$, and

- that an interval $[t_s, t_e]$ will intersect all intervals with grid points in the region bounded by the five lines $x = 0$, $y = 0$, $x = t_e$, $x + y = t_s$, $x + y = t_{\max}$ with $t_{\max}$ being the maximum of the endpoints of all intervals.

The spatial rendition is then divided into m strips: assume that $t_{\min}$ is the minimum of the startpoints of all intervals. Then the time domain between $t_{\min}$ and $t_{\max}$ is divided into m ranges $[t_0, t_1]$, $(t_1, t_2]$, $(t_2, t_3]$, ... , $(t_{m-1}, t_m]$ with $t_0 = t_{\min}$ and $t_m = t_{\max}$. This creates m strips with the k-th strip being bounded by the lines $x = 0$, $y = 0$, $x + y = t_{k-1}$ and $x + y = t_k$. Finally, each strip is divided by lines $x = t_1$, $x = t_2$, ... , $x = t_{m-1}$. This creates $\frac{m \cdot (m+1)}{2}$ fragmentation areas. Figure 4.13 shows the result of this process for $m = 4$ and the example that has been used throughout this chapter. The figure also demonstrates the way in which the resulting fragments are numbered.

For processing the join $R \bowtie_C Q$, a fragment R_k of R has to be joined with a certain set of fragments of Q. The members of this set are determined by k. The latter describes the position of the corresponding fragmentation area on the grid. This position indicates certain characteristics: for example, fragments[5] that are positioned in the upper part of the grid hold tuples whose interval timestamp is quite long. These tuples are likely to join with many others. In contrast, short-lived tuples are situated in the lower part of the grid. These are likely to join with only a few tuples. Intuitively, this means that fragments in the upper part need to be joined with more fragments than those located in the lower part of the grid. For the situation in figure 4.13 the necessary combination of fragments is shown in figure 4.14(a). For a more formal description of how to calculate these combinations see [Lu et al., 1994]. The example of figure 4.13, however, has several empty fragments which reduces its matrix of necessary joins to the one shown in figure 4.14(b).

Figure 4.15 shows the resulting search strategy. It is quite efficient in avoiding unnecessary comparisons in this case. This, however, is partially due to the fact that the example has only a few long-lived tuples which causes fragments $R_6, Q_6, R_7, Q_7, Q_9, R_{10}, Q_{10}$ to be empty. These are all fragments which are usually involved in many partial joins (see figure 4.14(a)). This reduces

[5] In this section, we use the term *fragment* for both, the actual subset of tuples *and* the fragmentation area of the two dimensional grid that determines which tuples become members of that subset. This is intended to simplify the description and to help the reader rather than to disturb him / her.

the number of necessary partial joins significantly (see figure 4.14(b)). Furthermore we note that tuples from one relation may be loaded sequentially but tuples from the second relation might require several accesses.

Alternatively, the partial joins can be processed in parallel. This, however, requires tuples of one relation to be replicated. Assume that this relation is Q. Consider now the example that we have used throughout the chapter. Regarding the matrix in figure 4.14(b) we note that R_5 and R_9 are to be joined with the same set of fragments of Q, namely Q_2, Q_3, Q_4, Q_5, Q_8. Thus we can pack R_5 and R_9 together into one 'super-fragment' which is to be joined with those fragments of Q. This avoids the need to provide both R_5 and R_9 with a copy each of those fragments, thus reducing tuple replication. Similarly, R_4 and R_8 can be packed together. Even R_1 and R_3 can form a super-fragment as R_1 has to join with a subset of R_3's Q-fragments. This groups the partial joins of figure 4.14(b) into three[6] 'super-partial joins'. This is illustrated in figure 4.16.

A major problem arises in the case that three or more relations participate in the join. Then, the structures depicted as matrices in figure 4.14 would have to be cubic or of higher order. This means that a huge number of fragment combinations have to be joined in order to compute the global result. For $m = 4$, for example, there are 70 partial joins for a global two-way join (see matrix in figure 4.14(a)). For $m = 4$ and a three-way join, however, there are already 528 partial joins to be computed.

The approach taken by Lu *et al.* is similar to many that have been used in the area of spatial join processing. Many spatial join algorithms are based on transforming an approximation of a spatial object into another domain, and performing filtering (e.g. in the form of partitioning) in the new domain [Patel and DeWitt, 1996]. Examples for such algorithms can be found in [Orenstein, 1986], [Orenstein and Manola, 1988] or [Becker et al., 1993].

[6]This allows the comparison of the resulting situation with those that are illustrated in figures 4.8 and 4.11.

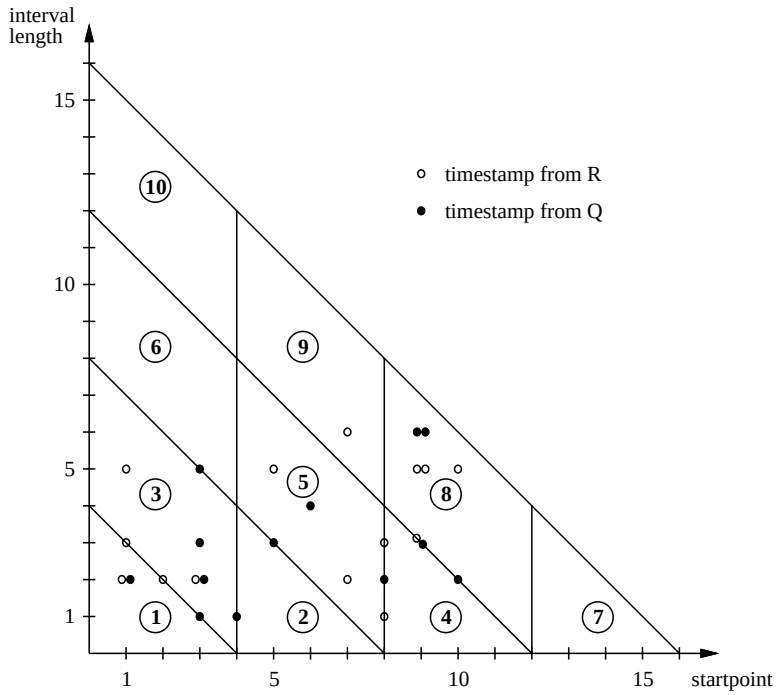


Figure 4.13: Spatial rendition and numbering of fragments for the example.

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}
R_1	●		●			●				●
R_2		●	●		●	●			●	●
R_3	●	●	●		●	●			●	●
R_4				●	●	●		●	●	●
R_5		●	●	●	●	●		●	●	●
R_6	●	●	●	●	●	●		●	●	●
R_7							●	●	●	●
R_8				●	●	●	●	●	●	●
R_9		●	●	●	●	●	●	●	●	●
R_{10}	●	●	●	●	●	●	●	●	●	●

(a) In general, for cases with $m = 4$.

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_8
R_1	●		●			
R_3	●	●	●		●	
R_4				●	●	●
R_5		●	●	●	●	●
R_8				●	●	●
R_9		●	●	●	●	●

(b) For the example of figure 4.15.

Figure 4.14: Partial joins that are to be computed for processing the spatially partitioned join.

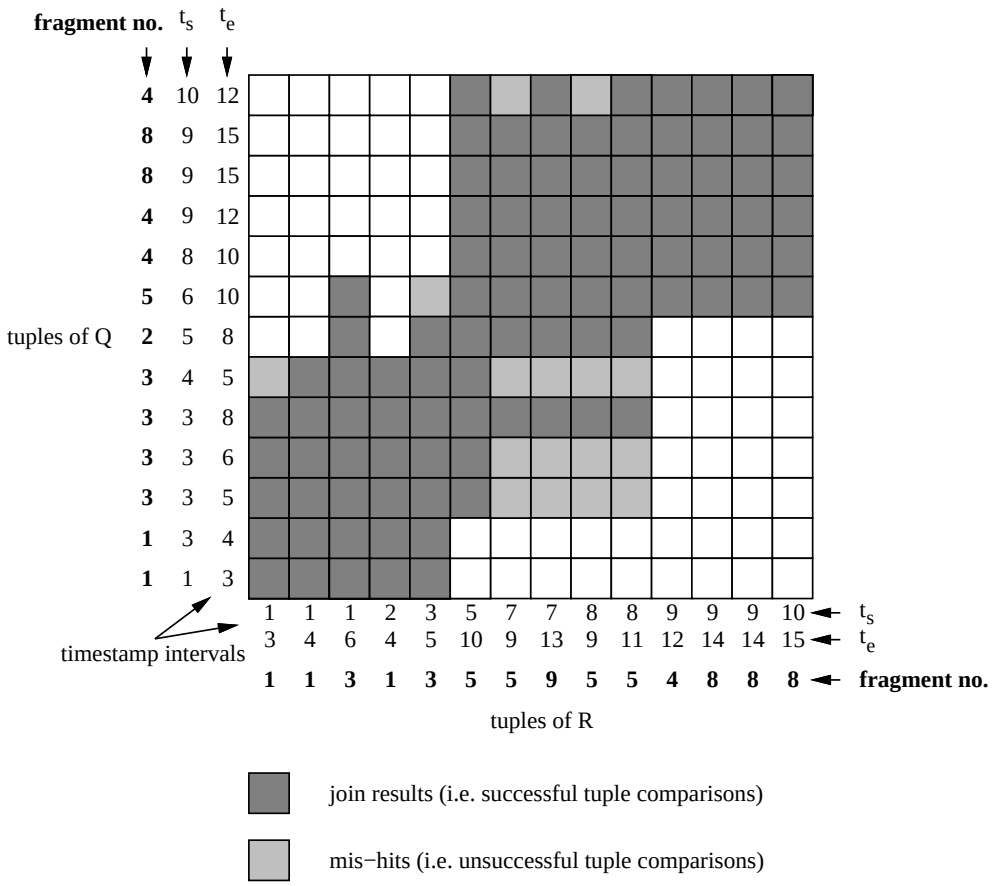


Figure 4.15: Search strategy for the spatially partitioned temporal join being processed sequentially. Partial joins are processed as nested loops.

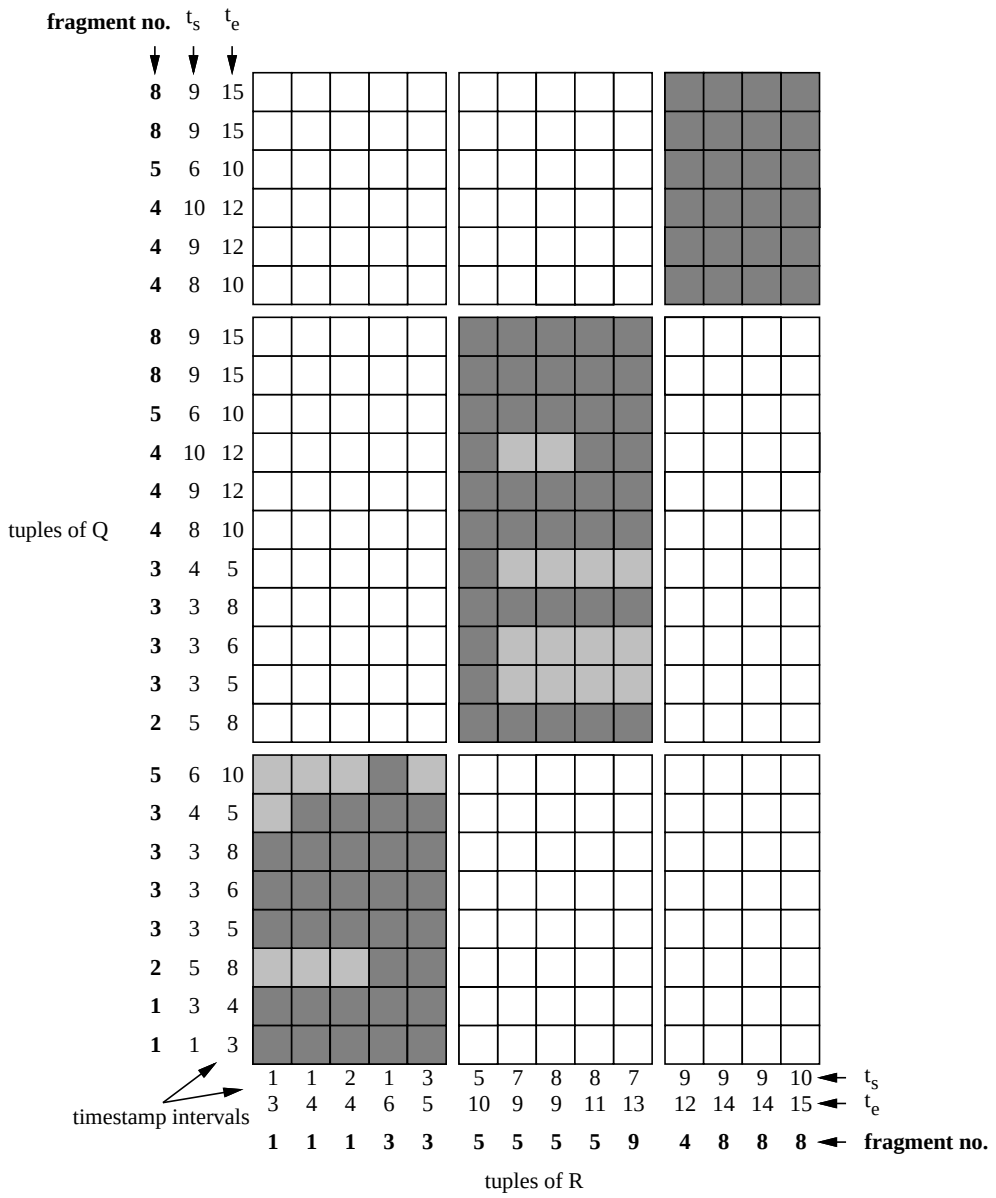


Figure 4.16: Search strategy for the spatially partitioned temporal join being processed in parallel. Partial joins are processed as nested loops.

4.5 A Short Summary

Having discussed several temporal join algorithms and techniques in the last two sections, we want to compare them in order to summarise their characteristics. We confirm our conclusions by reviewing the situations that were illustrated in figures 4.2, 4.4, 4.6, 4.8, 4.11, 4.15 and 4.16. Certainly, this considers only one specific example and therefore does not represent a thorough analysis. Nevertheless, it will give some insights into the characteristics of the algorithms.

First, we look at the sequential techniques. One significant performance characteristic then is the total number of tuple comparisons that are performed by the matching stage of the respective algorithm in order to get the result tuples. The nested-loop join is still the worst because it performs an exhaustive search. The difference to other methods is not that big as in the case of a comparable equi-join due to the higher selectivity factor of the temporal intersection in comparison to the equality predicate. The join index is still the clear winner as the join is virtually precomputed; the algorithm only composes the result. This leads to a minimal number of tuple comparisons[7]. Although the example suggests the sort-merge, the improved range partitioning and the spatial partitioning technique to be at equal levels, there is a significant difference which lies in the partitioning method by which they achieve their performance:

- If the relations are not sorted then the sort-merge might require a possibly expensive sorting stage prior to the matching stage.
- Range partitioning needs to replicate tuples and imposes therefore an additional overhead apart from the partitioning effort itself. When processed sequentially, the physical replication of tuples can be replaced by a logical one which might still cause replicated tuple comparisons.
- The spatial partitioning approach does not require replication. However, it creates fragments that have to be joined not only with one but several fragments from other relations that participate in the join. This problem is very severe if there are more than two relations involved.

Hence, the relatively low number of tuple comparisons in the matching stages of these algorithms are achieved by moving processing effort to a partitioning stage that precedes the matching. The partitioning differs widely and has its drawbacks:

[7] or *tuple accesses* in this particular case.

- *sorting* – if necessary – is generally more expensive than range or spatial partitioning,
- *range partitioning* needs to replicate tuples and imposes an overhead on the matching stage,
- *spatial partitioning* requires the matching stage to perform a large number of partial joins with tuples of one relation being accessed more than once.

Now, we turn to the parallel techniques. Although the fragment-and-replicate method performs equally with the others in terms of maximum tuple comparisons per partial join, there is the replication effort that is significantly higher in comparison to the other techniques. The partial joins of the fragment-and-replicate joins are better balanced than those based on symmetric partitioning. This is no coincidence as there are no constraints on partitioning. Therefore, it is easy to balance the fragments. The spatially partitioned join performs relatively well for the example. But remember that the example was particularly nice (see matrix in figure 4.14(b)). And again, in the case of three or more relations participating in the join, there might be a huge number of fragment combinations that have to be joined in order to compute the global result. This makes it also much more difficult to combine several partial joins to a kind of 'super-partial join', as in figure 4.16 where two of the original partial joins were respectively combined.

The extent of the replication overhead of the fragment-and-replicate approach and the problems of spatially partitioned n-way joins for $n \geq 3$ makes the range partitioning approach a possible compromise. In fact, one can expect its performance to be among the best. Furthermore it is fairly robust, e.g. it can be easily adapted to process n-way joins for $n \geq 3$. Finally, it is a straightforward adaption of the equi-join range (hash) partitioning method. Therefore many equi-join optimisation techniques can be applied, too. However, tuple replication has to be of concern, as a poor choice of ranges can increase the replication rate. Optimisation issues for range partitioned temporal joins are discussed in the following section.

4.6 Temporal-Specific Join Optimisation Issues

The discussions of temporal join techniques in sections 4.3 and 4.4 have shown that many performance related issues are not specific to temporal joins but similar to the case of the equi-join which is well investigated. Hence, many well-known optimisation techniques can be applied when processing temporal joins.

However, the necessary replication of tuples in the case of range partitioned joins is a problem which is specific to temporal joins. As outlined in section 4.4.1, it produces overheads of various types. *Therefore, controlling the extent of replication has to be a major task of the optimisation process for these joins.*

There are two problems that have to be solved:

1. The choice of the ranges, i.e. the partition points on the time line, is a very delicate one. Consider, for example, figure 4.17 which shows the scenario for our join example when an alternative set of partitioning ranges is used: more tuples have to be replicated, there are more mis-hits and the load balance is slightly worse in comparison to the situation in figure 4.11.

   The main part of this thesis will look at this problem. In chapter 5 we analyse the problem of finding partitioning ranges that minimise the total number of replicated tuples while preserving a certain maximum number of intervals per fragment. Afterwards, in chapters 6–10, the problem is tackled from a practical point of view: we develop and describe an optimisation process for choosing a 'good' partition for processing a temporal join and show how this process can be efficiently implemented.

2. Query optimisers usually estimate the sizes of join results. This information is used for taking optimisation decisions and sometimes also to provide the user with an estimate of the result size. This might help him / her to check if his / her query delivers the desired result. If the optimiser's estimate predicts a result size of 1 million rows, for example, while the user expects only a handful then there is a good reason for the user to assume that the query might have to be rewritten.

   The estimation of final and intermediate join results is already a challenging task because of the nonequi-character of temporal join conditions. It becomes even more difficult when tuples are replicated: the sizes of the join fragments and the partial join results cannot be computed by incorporating 'static' facts – i.e. facts that are known in advance from metadata

information such as relation size, number of different attribute values, distribution of these values (e.g. in form of histograms) etc. – but also on query evaluation parameters such as the replication of tuples imposed by the partition ranges.

In chapter 11 we will show a method to estimate temporal join selectivities. It is based on the same concepts as the optimisation of the partitioning process that was motivated in 1.

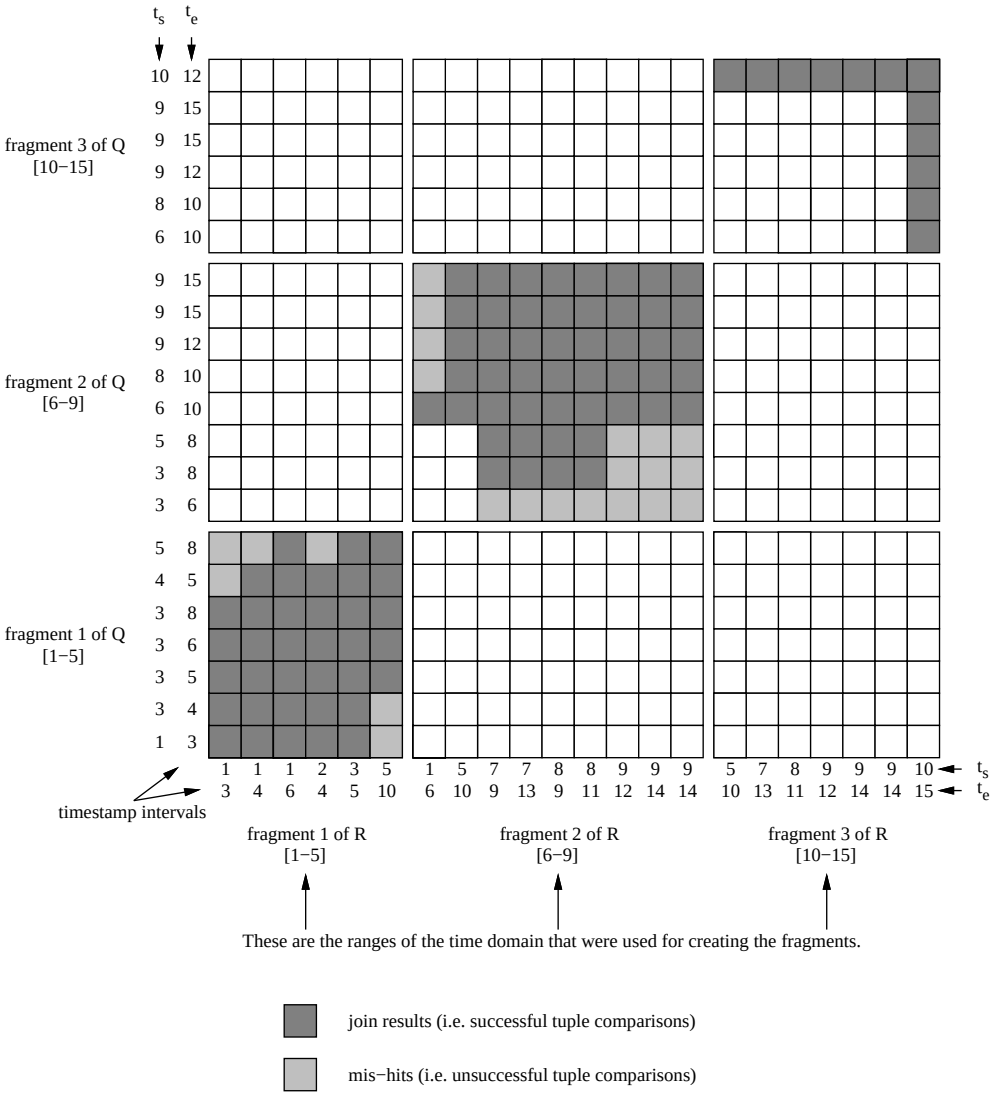


Figure 4.17: Search strategy for the improved (range) partitioned temporal join with different partitioning ranges (compare with figure 4.11).

Chapter 5

The Interval Partitioning Problem

5.1 Introduction

In this chapter, we analyse the problem of replication when partitioning a collection of intervals. As we have seen in section 4.4, this problem is relevant for the temporal join algorithms that are based on explicit partitioning of the data over the timestamp attribute.

The replication[1] of tuples can have an impact on the performance of the join algorithms in a variety of ways. The exact consequences for the join costs, however, can only be determined by considering characteristics that are algorithm- or hardware-specific:

- Sequential join algorithms are affected differently than parallel ones.
- Similarly, it will always be a matter of the underlying hardware, e.g. in the case of a parallel join:
  - Is it a shared-memory architecture that might not require the tuples to be replicated physically but only logically or is it a shared-nothing machine which will need the tuples to be physically replicated?
  - Are there any specific hardware components that improve certain communication patterns (e.g. fast broadcasts) that will be particularly useful for the respective algorithm?

This means that a partition that minimises the costs for one particular join algorithm running on one particular hardware might not do so well when applied to a different algorithm, possibly running on a different hardware. In

[1] Please remember that by this we mean a *logical* replication which might, but does not necessarily translate into a physical replication. Soo *et al.*'s join algorithm, for example, does not physically replicate tuples although several tuples are logically replicated because they are present in more than one fragment of the relation (see section 4.4.4).

this chapter, we want to avoid such situations and therefore we adopt a more general view. A partition is to be considered as optimal if it keeps fragments' sizes below a maximum value, while minimising the number of intervals that overlap the breakpoints. This does not necessarily minimise the join processing costs. Nevertheless, it will be beneficial *for every algorithm running on any piece of homogeneous*[2] *hardware* to have such an optimal partition.

The problem of finding such optimal partitions for interval data is called the *interval partitioning problem* (IP). This is a rather tricky problem as it not only requires to choose appropriate breakpoints but also makes it delicate to determine the number of breakpoints. In fact, we have encountered a dilemma: in order to reduce the number of overlapping intervals one has to reduce the number of breakpoints but in order to increase the number m of partial joins in (3.6) – e.g. for increasing the degree of parallelism – one has to increase the number of breakpoints. Thus there are two contrary effects associated with interval partitioning; note that the first one does not exist in the case of an equi-join. This is a manifestation of the *min-max dilemma* [Zhou, 1993]. We therefore need an additional input parameter. A useful constraint is to form fragments with less than a certain number of intervals. Such a number, for example, could be implied by the amount of memory or the disk space that is available.

In this chapter, we will formally investigate the complexity of IP, i.e. we will check whether a solution for IP can be found in polynomial time and – if so – we would like to have an algorithm that can compute this solution. The remainder of this chapter is structured like this: Section 5.2 introduces the notation that we are going to use throughout the chapter and the remaining parts of the thesis. In section 5.3, IP is defined formally. Section 5.4 looks at the search space of the problem. The main result will be that optimal partitions can be found within the set of interval endpoints. In section 5.5, we describe the algorithm `IP-opt` that computes solutions for IP in polynomial time. Finally, section 5.6 shows an alternative approach by relating IP to a graph-theoretic problem, namely the problem of sequential graph partitioning (SGP). This leads to a similar result as manifested by `IP-opt`. However, the connection between IP and graph partitioning (GP) might prove to be quite fruitful as the many complexity and algorithmic results about GP might be applied to variations of IP.

[2] A heterogeneous cluster of processors, e.g. a network of workstations, will require a specific load balancing approach that creates higher loads for more powerful nodes and smaller loads for less powerful ones.

5.2 Preliminaries

Before formally investigating the complexity of the interval partitioning problem, we introduce the notation that is used in the remainder of the chapter.

- In this chapter, we adopt a general view of the interval partitioning problem. Therefore we partition *collections of intervals* rather than *temporal relations*. Later, in the context of temporal join processing, we regard a temporal relation as a collection of intervals by neglecting all attribute values other than the interval data.

  To simplify the overall notation, we will use letters R and Q or expressions like $R \cup Q$ to refer to collections of intervals, having in mind that these originate in temporal relations R, Q or $R \cup Q$. Although being formally incorrect we think that this improves the readability and avoids confusing the reader. Similarly, we will use r and q to refer to tuples *and* intervals depending on the actual context.

  Formally, a collection of intervals is denoted by

  $$\langle r_1, \ldots, r_n \rangle$$

  The difference between a set and a collection is that an element can appear multiply in a collection but only once in a set. The cardinality is defined accordingly. In summary, this means that if $\langle r_1, \ldots, r_n \rangle$ is the collection of intervals resulting from a temporal relation R then

  $$|\langle r_1, \ldots, r_n \rangle| \; = \; n \; = \; |R|$$

- Intervals are defined over a certain domain. For our purposes, we can assume the set of integers to be the domain with the symbolic infimum and supremum values $-\infty$ and $+\infty$. We assume that there is a total ordering defined on the domain. As outlined in chapter 2, the set of integers is a reasonable representation of time for our purposes. If t is a timepoint then we refer to its predecessor by $t - 1$ and to its successor by $t + 1$.

- *Intervals* have the notations that have been introduced in section 2.2; the two that are relevant here are $[t_s, t_e]$ and $(t_s, t_e]$ in which t_s and t_e are timepoints / instants. In that sense, the intervals comprise all instances between and including the start- and endpoints in the case of $[t_s, t_e]$ and all instances between the start- and endpoints, excluding the start- but including the endpoint in case of $(t_s, t_e]$:

$$[t_s, t_e] = \{x : t_s \leq x \leq t_e\}$$
$$(t_s, t_e] = \{x : t_s < x \leq t_e\}$$

Please see section 2.2 for a more detailed discussion of this notation. We use the first type in the collection(s) of intervals that are to be partitioned and the second type for the partitioning ranges (see below).

- The *range* $T(R)$ of a collection R of intervals is the part of the domain covered by the intervals in R; it is formally defined as

$$\begin{aligned} T(R) &= \bigcup_{[t_s,t_e] \in R} [t_s, t_e] \\ &= \{t : t \text{ is contained in some } r \in R\} \end{aligned}$$

  We will refer to the minimum of $T(R)$ as $t_{\min}$ and to the maximum as $t_{\max}$. The collection to which $t_{\min}$ and $t_{\max}$ refer will always be obvious from the context.

- The *span* $L(R)$ of a collection R of intervals is defined by

$$L(R) = [t_{\min}, t_{\max}]$$

  The span may contain parts of the domain that are not covered by any interval and are therefore not included in the range $T(R)$.

- The sets of interval startpoints, $S(R)$, and endpoints, $E(R)$, are defined by

$$\begin{aligned} S(R) &= \{t_s : \exists t_e \in T(R) \text{ such that } [t_s, t_e] \in R\} \\ E(R) &= \{t_e : \exists t_s \in T(R) \text{ such that } [t_s, t_e] \in R\} \end{aligned}$$

- A *partition* P is an ordered[3] set

$$\{p_1, \ldots, p_{m-1}\}$$

  of *breakpoints* that divide the span and range into m *segments* or *partition ranges* $(p_{k-1}, p_k]$ for $k = 1, \ldots, m$ with p_0 and p_m being defined appropriately: suitable values, for example, are

$$t_{\min} - 1 \qquad \text{or} \qquad -\infty$$

[3] As a slight abuse of set theory, we require some sets to be ordered. This is for naming purposes only and simplifies the notation in general. Therefore we refrain from introducing separate brackets and operator symbols to distinguish between conventional and ordered sets as this would probably confuse the reader more than our slightly abusive notation.

for p_0 and

$$t_{\max} \qquad \text{or} \qquad +\infty$$

for p_m. We do not consider p_0 and p_m to be part of a partition P for two reasons: (a) they can be the same for all possible partitions and (b) they do not actually influence the performance characteristics of P as we will see later. The difficult choice is to determine the $p_1, \ldots, p_{m-1}$ and this is our main concern.

- In order to calculate the impacts of a partition, we use the following functions which are assumed to be defined on the entire set of integers although non-zero values only occur for $t \in L(R)$:

$$\begin{array}{lcll} s_R(t) & = & \text{number of intervals in } R \text{ that start at point } t & = & |\langle r \in R : r.t_s = t \rangle| \\ e_R(t) & = & \text{number of intervals in } R \text{ that end at point } t & = & |\langle r \in R : r.t_e = t \rangle| \\ i_R(t) & = & \text{number of intervals in } R \text{ that include point } t & = & |\langle r \in R : r.t_s \leq t \leq r.t_e \rangle| \\ o_R(t) & = & \text{number of intervals in } R \text{ that overlap point } t & = & |\langle r \in R : r.t_s \leq t < r.t_e \rangle| \\ & & \text{(i.e. they include } t \text{ but do not end at } t\text{)} & & \end{array}$$

The functions s_R, e_R, i_R, o_R will prove to be quite useful for demonstrating a variety of properties. Actually, each of them can be expressed by using two of the others. Their relationships can be derived from the three basic properties that are obvious from the definition of s_R, e_R, i_R and o_R:

$$o_R(t) = i_R(t) - e_R(t) \tag{5.1}$$

$$i_R(t) = s_R(t) + o_R(t-1) \tag{5.2}$$

and

$$s_R(t) = e_R(t) = i_R(t) = o_R(t) = 0 \qquad \text{for } t < t_{\min} \text{ and } t > t_{\max} \tag{5.3}$$

Equation (5.1) reflects the fact that the intervals that overlap t are those that include t apart from those that end at t. Equation (5.2) considers that intervals that include t must either start at t – these are counted by $s_R(t)$ – or have started at $t-1$ or before without ending at $t-1$ – these are counted by $o_R(t-1)$. Finally, (5.3) is rather trivial as it indicates that nothing goes on outside the range $T(R)$.

If at least two of the functions s_R, e_R, i_R, o_R are known then we can compute the missing ones by using equations (5.1), (5.2) and (5.3). Figure 5.1 shows the equations that can be derived. The case of having the values $s_R(t)$ and $e_R(t)$ – this corresponds to figure 5.1(a) – is a little bit tricky because the equations are recursive: Replacing $i_R(t)$ in (5.1) by using (5.2) delivers

$$o_R(t) = o_R(t-1) + s_R(t) - e_R(t) \tag{5.4}$$

which works out to be

$$o_R(t_{\min}) = s_R(t_{\min}) - e_R(t_{\min}) \tag{5.5}$$

as we can derive $o_R(t_{\min} - 1) = 0$ from (5.3). Using (5.5) as a starting point, subsequent values can be calculated by applying (5.2) or (5.4) respectively.

The remaining equations shown in figure 5.1 result from algebraic transformations and combinations of equations (5.1) and (5.2).

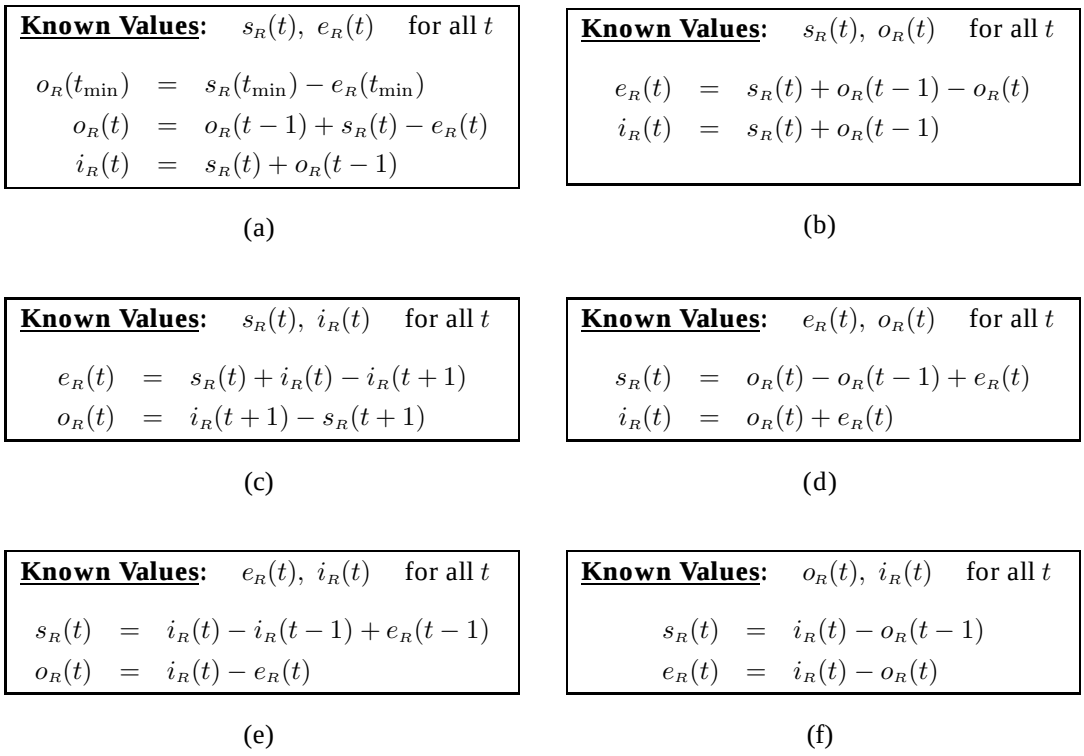


Figure 5.1: Relationships between s_R, e_R, o_R and i_R.

5.3 Problem Definition

We can now formally define the interval partitioning (IP) problem. An instance of IP consists of a collection R of intervals and a limit X for the maximum number of intervals allowed in a partition fragment. The reason why we have to allow collections rather than sets is the following: In a temporal relation, for example, two tuples may be distinct yet but can have the same timestamp interval. In the partitioning process it might be necessary to count every tuple but actually neglect all attribute values apart from the timestamp. Intervals that originate from one or more temporal relations can therefore appear more than once and it is important to know how many times.

Figure 5.2 shows an example of such a situation using a simple, uniform partition of the time domain. There are twenty intervals – represented as bold bars – and a uniform partition of the time line that goes from 0 to 20. The breakpoints – 5, 10 and 15 in this case – are shown by vertical lines. The breakpoints themselves belong to the respective left fragment[4]. The figure then gives the resulting loads of the fragments in circles which add up to a total of 39. This means that the partitioning process caused 19 overlaps, i.e. 19 times intervals cross breakpoint lines.

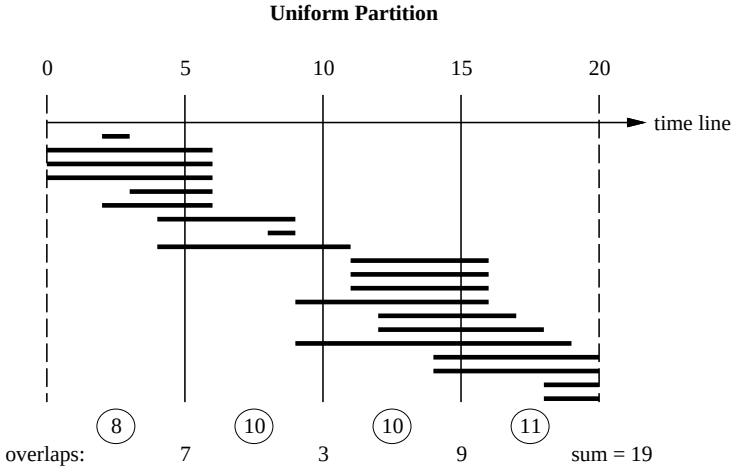


Figure 5.2: A collection of intervals that has been uniformly partitioned.

The goal is now to find an integer m and a partition $P = \{p_1, \ldots, p_{m-1}\}$ of the span $L(R)$. m can hold any suitable value. The intervals of R are partitioned such that $r \in R$ is put into a fragment R_k if and only if r intersects with the

[4]Therefore, an interval falls into both, the left and the right fragment if it starts at the breakpoint.

partition range $(p_{k-1}, p_k]$ that corresponds to R_k ($k = 1, \ldots, m$). There are two constraints for an optimal P:

1. The total numbers of intervals that overlap the partition points p_1, ..., p_{m-1} is minimal.
2. No R_k can hold more than X intervals ($k = 1, \ldots, m$).

The definition is summarised in figure 5.3.

Definition: Interval Partitioning – IP

Instance: $ip(R, X)$

- A collection R of intervals $\langle r_1, \ldots, r_N \rangle$,
- a positive number X.

Question:
Is there a partition $P = \{p_1, \ldots, p_{m-1}\}$ of R with $p_{k-1} < p_k$ for $k = 2, 3, \ldots, m-1$ that minimises

$$\sum_{p \in P} o_R(p) \tag{5.6}$$

such that

$$|R_k| \leq X \tag{5.7}$$

for all $k = 1, \ldots, m$ where

$$R_k = \langle r \in R : [r.t_s, r.t_e] \cap (p_{k-1}, p_k] \neq \emptyset \rangle \quad ?$$

Figure 5.3: Definition of IP

Please note the following: the number of intervals within a fragment R_k can be calculated from the value of the functions s_R and o_R. Intervals that intersect with the partition range $(p_{k-1}, p_k]$ either

- start within the partition range, i.e. $p_{k-1} < r.t_s \leq p_k$; the number of intervals with this property is
$$\sum_{t=p_{k-1}+1}^{p_k} s_R(t)$$
- or they have started at p_{k-1} or before and overlap p_{k-1}; the number of intervals that have this property is
$$o_R(p_{k-1})$$

Consequently, the number of intervals falling into R_k is the sum of these two figures. This is summarised in the following

Lemma 1 *Given a collection R of intervals and a partition $P = \{p_1, \ldots, p_{m-1}\}$ is given by the number of intervals falling into a fragment $R_k = \langle r \in R : [r.t_s, r.t_e] \cap (p_{i-1}, p_i] \neq \emptyset \rangle$ by the equation*

$$|R_k| \;=\; o_R(p_{k-1}) + \sum_{t=p_{k-1}+1}^{p_k} s_R(t) \tag{5.8}$$

Proof:

See explanation above. □

5.4 Search Space

Looking at the scenario in figure 5.2, we can see that one major problem of the uniform partition is that its breakpoints go through the interior part of a large number of intervals. In fact, there is no breakpoint which coincides with an interval start- or endpoint. Intuitively, we can move the breakpoints slightly to the left or right to the next point on which one or more 'broken' intervals begin or end. This measure possibly reduces but at least does not increase the number of overlaps. This observation suggests that an optimal partition should have its breakpoints within $S(R) \cup E(R)$. The remainder of this section shows that this is in fact true. We will even show that optimal partitions can be found within $E(R)$.

Assume that there is an optimal partition $P = \{p_1, \ldots, p_{m-1}\}$ for some instance *ip*(R, X) of IP. Furthermore assume that P has a breakpoint p_k that is not an endpoint of some interval in R, i.e. $p_k \notin E(R)$. We will show that P can be converted into a partition that has only breakpoints within $E(R)$ and is still optimal in the sense of IP. To that end, breakpoints such as p_k can be moved to the nearest endpoint to the left. See figure 5.4 for an example that shows the benefit of this measure. First, we show that such an endpoint always exists.

Lemma 2 *Let ip(R, X) be an instance of IP and $P = \{p_1, \ldots, p_{m-1}\}$ with $p_{k-1} < p_k$ for $k = 2, 3, \ldots, m-1$ an optimal partition according to IP. Then there is always an $e \in E(R)$ such that*

$$t_{\min} \leq e \leq p_1$$

Proof:

We prove the lemma by contradiction.
Assume that there is an optimal partition P such that there is no $e \in E(R)$ with $t_{\min} \leq e \leq p_1$, i.e. no interval ends within the first partition range and thus also intersects with the second one. This means that every interval in the first fragment R_1 – and there must be some because $P \subseteq T(R)$ – must also be in the second fragment R_2, i.e.

$$R_1 \subseteq R_2 \quad \text{with} \quad R_1 \neq \emptyset \qquad (*)$$

This means also that

$$o_R(p_1) > 0 \qquad (**)$$

Now, consider the partition $P' = \{p_2, \ldots, p_{m-1}\}$. It creates the fragments

$$\begin{aligned} R_1 \cup R_2 &\overset{(*)}{=} R_2 \\ &\quad R_3 \\ &\quad R_4 \\ &\quad \ldots \\ &\quad R_m \end{aligned}$$

with $|R_k| \leq X$ for all $k = 2, \ldots, m$. Thus P' satisfies constraint (5.7) of IP. Furthermore we have

$$\sum_{p' \in P'} o_R(p') = \sum_{k=2}^{m-1} o_R(p_k) \overset{(**)}{<} \sum_{k=1}^{m-1} o_R(p_k) = \sum_{p \in P} o_R(p)$$

Thus P is not optimal in the sense of IP. This contradicts the initial assumption and therefore the lemma holds. □

Now, we will refer to the nearest endpoint to the left of any point t by *left*(t). It is defined as

$$\textit{left}(t) = \max\{e \in E(R) \cup \{t_{\min}\} : e \leq t\}$$

Because of lemma 2 this is reduced to

$$\textit{left}(p) = \max\{e \in E(R) : e \leq p\} \qquad (5.9)$$

if p is a breakpoint of an optimal partition.

The next lemma will show how an optimal partition $P \subseteq T(R)$ can be subsequently converted into a partition $\bar{P}$ which is (a) optimal for the same instance of IP and (b) consists only of breakpoints that are endpoints of intervals of R. The lemma assumes that the leftmost breakpoint $p_k \in P$ which is not an endpoint is converted first.

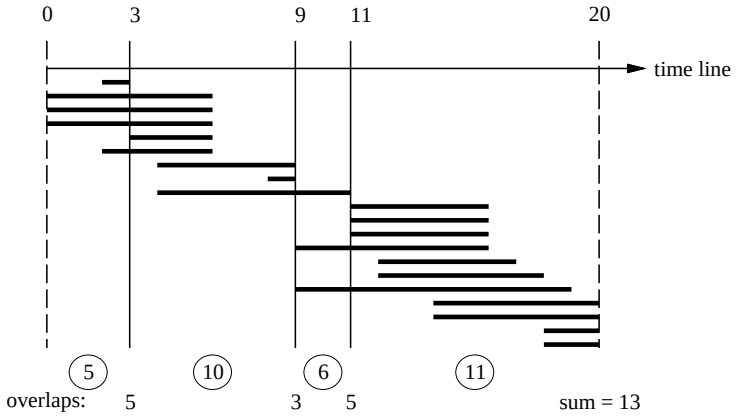


Figure 5.4: Moving breakpoints to the nearest endpoints to the left in case of the example of figure 5.2.

Lemma 3 *If there is an instance ip(R, X) of IP and a partition*

$$P \;=\; \{p_1, \ldots, p_{k-1}, p_k, p_{k+1}, \ldots, p_{m-1}\} \;\subseteq\; T(R)$$

with $p_{k-1} < p_k$ for all $k = 2, \ldots, m-1$ that satisfies the constraints (5.6) and (5.7) of IP then the partition

$$\bar{P} \;=\; \{p_1, \ldots, p_{k-1}, \mathit{left}(p_k), p_{k+1}, \ldots, p_{m-1}\}$$

for any $p_k \in P$ with $\{p_1, \ldots, p_{k-1}\} \subseteq E(R)$ also satisfies these constraints.

Proof:

If $p_k \in E(R)$ then

$$p_k = \mathit{left}(p_k)$$

according to (5.9). Thus $\bar{P} = P$ and the lemma holds.

In the following we assume that $p_k \notin E(R)$ and therefore that $p_k > \mathit{left}(p_k)$. Because of $\{p_1, \ldots, p_{k-1}\} \subseteq E(R)$ we also know that $p_{k-1} \leq \mathit{left}(p_k)$.

From $p_k \notin E(R)$ and the definition (5.9) of *left* we can conclude that

$$e_{\scriptscriptstyle R}(t) = 0 \quad \text{for all } \mathit{left}(p_k) < t \leq p_k \qquad (*)$$

From (5.1) and (5.2) we get

$$o_{\scriptscriptstyle R}(t-1) \;=\; o_{\scriptscriptstyle R}(t) - s_{\scriptscriptstyle R}(t) + e_{\scriptscriptstyle R}(t)$$

for all t in the domain. But for the t with $\mathit{left}(p_k) < t \leq p_k$ this works out to be

$$o_{\scriptscriptstyle R}(t-1) \;=\; o_{\scriptscriptstyle R}(t) - s_{\scriptscriptstyle R}(t) \qquad (**)$$

because of $(*)$. But $(**)$ implies that

$$\begin{aligned} o_R(\mathit{left}(p_k)) &= o_R(\mathit{left}(p_k)+1) - s_R(\mathit{left}(p_k)+1) \\ &= o_R(\mathit{left}(p_k)+2) - s_R(\mathit{left}(p_k)+1) - s_R(\mathit{left}(p_k)+2) \\ &= \cdots \\ &= o_R(\mathit{left}(p_k)+\Delta p) - \sum_{j=1}^{\Delta p} s_R(\mathit{left}(p_k)+j) \end{aligned}$$

with $\mathit{left}(p_k) < \mathit{left}(p_k) + \Delta p \leq p_k$. In particular, i.e. for $\mathit{left}(p_k) + \Delta p = p_k$, this means that

$$o_R(\mathit{left}(p_k)) = o_R(p_k) - \sum_{t=\mathit{left}(p_k)+1}^{p_k} s_R(t) \qquad (***)$$

and we can conclude that

$$o_R(\mathit{left}(p_k)) \leq o_R(p_k)$$

But this means that

$$\sum_{\bar{p}\in\bar{P}} o_R(\bar{p}) \leq \sum_{p\in P} o_R(p)$$

i.e. that $\bar{P}$ satisfies constraint (5.6)[5].

Now we look at constraint (5.7). Let us refer to the fragments created by P as $R_1, \ldots, R_m$ and to those created by $\bar{P}$ as $\bar{R}_1, \ldots, \bar{R}_m$. Trivially, it is

$$\bar{R}_j = R_j$$

for all $j = 1, \ldots, k-1, k+2, \ldots, m-1$ and therefore also

$$|\bar{R}_j| = |R_j| \leq X$$

for those j as P satisfies (5.7). Thus we need to look at the sizes of the fragments $\bar{R}_k$ and $\bar{R}_{k+1}$. These can be derived from the sizes of R_k and R_{k+1} in the following way using the lemma 1:

$$\begin{aligned} |R_k| &= o_R(p_{k-1}) + \sum_{t=p_{k-1}+1}^{p_k} s_R(t) \\ &= o_R(p_{k-1}) + \sum_{t=p_{k-1}+1}^{\mathit{left}(p_k)} s_R(t) + \sum_{t=\mathit{left}(p_k)+1}^{p_k} s_R(t) \\ &\geq o_R(p_{k-1}) + \sum_{t=p_{k-1}+1}^{\mathit{left}(p_k)} s_R(t) \\ &= |\bar{R}_k| \end{aligned}$$

[5]Later, after having proved that $\bar{P}$ also satisfies the second constraint of IP, we can conclude that the two sums are equal; otherwise P would not be a minimising partition as required by the lemma.

Thus it is $|\bar{R}_k| \leq |R_k| \leq X$. Again, we can conclude later that $|\bar{R}_k| = |R_k|$ because otherwise P would not be a minimising partition. Now, we look at the $(k+1)$-th fragments:

$$\begin{aligned}
|R_{k+1}| &= o_R(p_k) + \sum_{t=p_k+1}^{p_{k+1}} s_R(t) \\
&\overset{(***)}{=} o_R(\mathit{left}(p_k)) + \sum_{t=\mathit{left}(p_k)+1}^{p_k} s_R(t) + \sum_{t=p_k+1}^{p_{k+1}} s_R(t) \\
&= o_R(\mathit{left}(p_k)) + \sum_{t=\mathit{left}(p_k)+1}^{p_{k+1}} s_R(t) \\
&= |\bar{R}_{k+1}|
\end{aligned}$$

Thus it is $|\bar{R}_{k+1}| = |R_{k+1}| \leq X$. Therefore $\bar{P}$ also satisfies constraint (5.7). □

Lemma 3 can now be used to subsequently convert an optimal partition P that has breakpoints that are no interval endpoints into one that is at least as good and that has only breakpoints within $E(R)$. Knowing this, the search for partitions that satisfy the IP-constraints can be restricted to a search within $E(R)$. This is expressed in the following

Theorem 1 *If there is an instance $ip(R, X)$ of IP and a partition $P \subseteq T(R)$ that satisfies the constraints of IP then there is also a partition $\bar{P} \subseteq E(R)$ that satisfies these constraints.*

Proof:

Apply the transformation discussed in lemma 3 subsequently to P until there are no more breakpoints that are not members of $E(R)$. The result is a partition $\bar{P} \subseteq E(R)$. □

The significance of this theorem is that we can now restrict the search for an optimal partition to the set of interval endpoints which is finite:

$$|E(R)| \leq |R|$$

This also relates the complexity of the problem to the number of intervals and not to the length of the span.

5.5 Optimal Partitioning

In this section, we will give an algorithm that computes an optimal partition for an instance of IP if such a partition exists. We recall that no optimal par-

tition exists in the case that there is no partition that satisfies constraint (5.7) of IP. Section 5.5.1 describes the algorithm `IP-opt`. Section 5.5.2 shows how `IP-opt` works for the example of figures 5.2 and 5.4. Finally, in section 5.5.3, we prove that `IP-opt` is correct.

5.5.1 Algorithm for Optimal Partitioning

A dynamic programming approach can be used for computing an optimal solution for IP if there is such. An optimal partition can be found amongst the set of endpoints as we have seen from theorem 1 in the previous section. We will refer to the elements in the set of endpoints as $q_1, \ldots, q_n$ with $q_i < q_{i+1}$ for all $i = 1, \ldots, n-1$.

We first describe the algorithm `IP-opt` informally. It starts with q_1 and goes through to q_n. For each q_i it will hold the necessary information for an optimal partition for the span ending at q_i, i.e. for the segment $[t_{\min}, q_i]$, with all intervals in R intersecting with the partial span being considered. The algorithm computes two items of information for each q_i:

- $c(q_i) =$ the cumulative total number of overlaps for the selected optimal partition up to q_i,
- $\mathit{pred}\,(q_i) = q_j$ if q_j is the previous breakpoint that led to this minimum.

A dummy point q_0 with $q_0 < q_1$ is used to provide a value for $\mathit{pred}\,(q_1)$. This is not actually necessary but improves the readability of the algorithm and later the correctness proof. The expression $\mathit{load}\,(q_j, q_i)$ with $q_j < q_i$ gives the number of intervals of R that fall into a fragment with partition range $(q_j, q_i]$, i.e.

$$\begin{aligned} \mathit{load}\,(q_j, q_i) &= |\langle r \in R : [r.t_s, r.t_e] \cap (q_j, q_i] \neq \emptyset \rangle| \\ &= o_R(q_j) + \sum_{t=q_j+1}^{q_i} s_R(t) \qquad \text{see lemma 1} \end{aligned} \tag{5.10}$$

Finally, the optimal partition – if there is one – is given by the sequence

$$\mathit{pred}\,(q_n),\ \mathit{pred}\,(\mathit{pred}\,(q_n)),\ \ldots$$

which ends when it produces q_0 as a breakpoint.

There is no optimal partition if – for any q_i – there is no q_j with $j < i$ such that the number of intervals intersecting with the segment $(q_j, q_i]$, i.e. $\mathit{load}\,(q_j, q_i)$, is less than the maximum load of X intervals. The loads involving

the dummy point are defined as

$$\begin{aligned} \textit{load}\,(q_0, q_1) &= 0 \\ \textit{load}\,(q_0, q_i) &= \textit{load}\,(q_1, q_i) \qquad \text{for } i = 2, \ldots, n \end{aligned}$$

The algorithm then looks as shown in figure 5.5.

Algorithm `IP-opt`

- $c(q_0) = 0$
- **for** $i = 1$ **to** n **do**
  - $J = \{j : 0 \leq j < i \ \wedge \ \textit{load}\,(q_j, q_i) \leq X\}$
  - **if** $J = \emptyset$ **then**
    output "No optimal partition." ; stop.
  - $c(q_i) = \min_{j \in J}\{o_R(q_j) \ + \ c(q_j)\}$
    Let $\textit{pred}\,(q_i) = q_j$ for the minimising q_j.
    If there is more than one qualifying q_j then choose the smallest.

  /* output of breakpoints in descending order */
- $p = q_n$
- **while** $p \geq q_1$ **do**
  - $p = \textit{pred}\,(p)$
  - output p

Figure 5.5: The algorithm `IP-opt` for computing an optimal partition for an instance of IP.

The run time complexity is $O(n^2)$ as the min-function is $O(n)$. Computing the values for the *load*-function is $O(n^2)$ and can be done beforehand. Similarly, computing the values for o_R is $O(n)$.

5.5.2 Example

Table 5.1 shows the values within `IP-opt` for the example shown in figures 5.2 and 5.4. The set of endpoints comprises the values found in the second column of the table; it is assumed that $q_0 = -1$. The table is calculated by `IP-opt` starting at the top with $q_1 = 3$ and proceeding to $q_{10} = 20$. The third column contains the value of $\textit{pred}\,(q_i)$ that leads to the minimum partition costs $c(q_i)$

that are shown in the fourth column. An optimal partition for the instance of IP can be derived from the table by the sequence

$$\mathit{pred}\,(20) = 17, \quad \mathit{pred}^{\,2}(20) = 9, \quad \mathit{pred}^{\,3}(20) = 6$$

We note that $c(20)$ corresponds to the minimum number of overlaps for this instance of IP. The optimal partition derived from the table is shown in figure 5.6.

i	q_i	$\mathit{pred}\,(q_i)$	$c(q_i)$
1	3	-1	0
2	6	-1	0
3	8	-1	0
4	9	6	2
5	11	6	2
6	16	9	5
7	17	9	5
8	18	17	9
9	19	17	9
10	20	17	9

Table 5.1: Values within `IP-opt` for the example in figures 5.2 and 5.4.

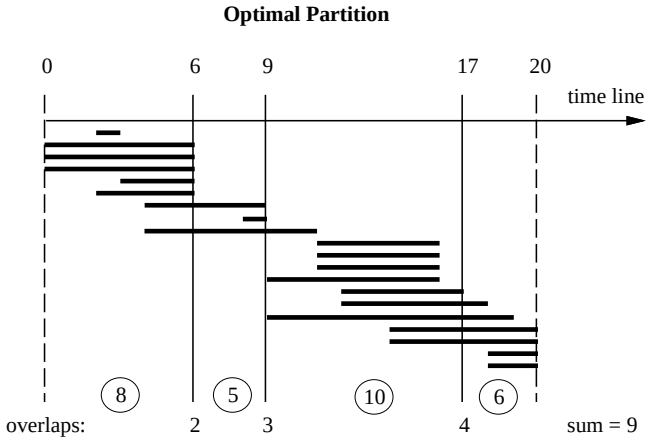


Figure 5.6: Optimal partition for the intervals of figures 5.2 and 5.4.

5.5.3 Correctness

In this section, we show that `IP-opt` is correct by proving the following

Theorem 2 *The algorithm* `IP-opt` *delivers an optimal partition for an instance of IP if there is a partition that satisfies (5.7).*

Proof:

The proof is by induction over the loop variable i for which

$$c(q_i) \;=\; \sum_{k=1}^{|P_i|} o_R(\mathit{pred}^{\,k}(q_i)) \tag{5.11}$$

is the invariant of the for-loop in `IP-opt` where

$$\mathit{pred}^{\,k}(q_i) \;=\; \underbrace{\mathit{pred}\,(\mathit{pred}\,(\ldots \mathit{pred}}_{k}(q_i)\ldots))$$

and

$$P_i \;=\; \{\,\mathit{pred}^{\,k}(q_i) :\; k \geq 1 \;\wedge\; \mathit{pred}^{\,k}(q_i) \geq q_1\}$$

Using the facts

- that $c(q_i)$ is always forced to be minimal through the *min* function
- and that P_i is the optimal partition for the segment $[t_{\min}, q_i]$

we can conclude that $c(q_n)$ is the number of overlaps in an optimal partition as specified by IP and that an optimal partition is given by P_n.

If there is no partition that satisfies constraint (5.7) then there must be some q_i such that $\mathit{load}\,(q_j, q_i) > X$ for all $j < i$. This causes the set J to be empty and thus the algorithm to report this fact and then to stop. In the remainder of the proof we assume that there is an optimal partition and consequently that J is non-empty for all $i = 1, \ldots, n$.

Base case: $i = 1$

The algorithm provides

$$J = \{0\} \;\Rightarrow\; c(q_1) = \min\{0\} = 0 \quad (*)$$

with $\mathit{pred}\,(q_1) = q_0$. Thus $P_1 = \emptyset$ which causes the sum in (5.11) to evaluate to 0 which matches with $(*)$. Thus the assumption holds.

Hypothesis: Assume that (5.11) holds for all $i \leq x \quad (**)$.

Inductive Step: $i = x + 1$

Let $q_j = \mathit{pred}\,(q_i) = \mathit{pred}\,(q_{x+1})$ be the minimising point with $j < i = x + 1$

as guaranteed by the algorithm. Then

$$\begin{aligned} c(q_{x+1}) &= o_R(q_j) + c(q_j) \\ &\overset{(**)}{=} o_R(q_j) + \sum_{k=1}^{|P_j|} o_R(\mathit{pred}^k(q_j)) \\ &= o_R(\mathit{pred}(q_{x+1})) + \sum_{k=2}^{|P_j|+1} o_R(\mathit{pred}^k(q_{x+1})) \\ &= \sum_{k=1}^{|P_{x+1}|} o_R(\mathit{pred}^k(q_{x+1})) \end{aligned}$$

Therefore (5.11) holds for $i = x + 1$.

□

5.6 Alternative: Reducing IP to a Graph-Theoretic Problem

The instances of the interval partitioning problem can be reduced to instances of a similar graph-theoretic problem, namely the problem of sequential graph partitioning (SGP). SGP was tackled at the beginning of the 1970s when people where looking for optimal code segmentations and paginations. A polynomial-time algorithm that computes optimal solutions for SGP was presented in [Kernighan, 1971]. For the purpose of interval partitioning we have to use a minor variation of SGP which does not change its complexity.

Being able to map instances of IP to instances of graph partitioning (GP) has another advantage: GP and its variations are well investigated and there is a variety of algorithmic and complexity results available. GP has proven to be very sensitive, even to minor changes in the problem constraints. Arbitrary GP, for example, is NP-complete [Hyafil and Rivest, 1973] whereas SGP is polynomial. Variations of IP probably behave similarly. Reducing them to a GP problem will enable us to benefit from a huge collection of algorithms and complexity results that have already been obtained.

The remainder of this section is structured like this: Section 5.6.1 introduces SGP in the form in which it is required for solving IP. Then, in section 5.6.2, we show how instances of IP can be reduced to instances of SGP. This step is essential as it opens the way towards finding optimal solutions for IP. Section 5.6.3

gives an example by reducing the example of figures 5.2, 5.4 and 5.6 to an instance of SGP. Section 5.6.4 formally proves that the reduction that we derived is correct. In section 5.6.5 the algorithm `SGP-opt` is presented which computes optimal partitions for instances of SGP. Section 5.6.6 gives an example that shows how `SGP-opt` works. Finally, we derive the runtime complexity of `SGP-opt` in section 5.6.7.

5.6.1 Sequential Graph Partitioning

An instance of the sequential graph partitioning (SGP) problem consists of a graph $G = (V, A)$ and a non-negative integer X that is used as the limit for each partition fragment. There is a total ordering $\prec$ defined on the vertex set $V = \{v_1, \ldots, v_N\}$ such that $v_i \prec v_j$ for $i < j$. A weight $w(v)$ is assigned to each vertex $v \in V$ and a length $l(v_i, v_j)$ to each edge $(v_i, v_j) \in A$. The goal is to partition V into subsets $V_1, \ldots, V_m$ with each V_k holding only consecutive vertices – thus the name *sequential* graph partitioning in contrast to traditional graph partitioning which is NP-complete [Hyafil and Rivest, 1973]. The optimality constraints are that partitioning

- minimises the sum of the lengths of the edges that start and end in different partition fragments and
- leaves the 'weight' of each fragment V_k less than or equal to X.

The 'weight' of a fragment V_k is usually the sum of the weights $w(v)$ for the $v \in V_k$. For our purpose, however, we have to add the lengths of incoming edges to this weight. This is a minor change to the problem tackled in [Kernighan, 1971] and does not change the complexity of the problem. Figure 5.7 summarises the definition of the SGP problem.

In section 5.6.3, we will give an example of SGP, actually the instance of SGP that results from reducing the IP example of figures 5.2, 5.4 and 5.6 to an SGP problem.

5.6.2 Reducing IP to SGP

Reducing instances of IP to instances of SGP is based on the following observations:

1. Theorem 1 showed that an optimal partition for an instance of IP can be found within the set $E(R)$ of interval endpoints. Therefore we can

Definition: Sequential Graph Partitioning – SGP

Instance: *sgp*(G, X)

- An undirected graph $G = (V, A)$ with a total ordering $\prec$ defined on the vertex set

$$V = \{v_1, \ldots, v_N\}$$

such that $v_i \prec v_j$ for all $i < j$. $A \subseteq V \times V$ is the set of edges. For convenience we assume an additional dummy vertex v_0 and that $i < j$ for all $(v_i, v_j) \in A$.

- A function $w : V \to \{0, 1, 2, \ldots\}$ assigning a weight $w(v)$ to each vertex $v \in V$,
- a function $l : A \to \{0, 1, 2, \ldots\}$ assigning a length $l(v_i, v_j)$ to each edge $(v_i, v_j) \in A$ and
- a positive number X.

Question:

Is there a partition of V into subsets $V_1, \ldots, V_m$ of consecutive vertices, i.e.

$$V_k = \{v_{(\tilde{p}_{k-1}+1)}, \ldots, v_{\tilde{p}_k}\}$$

imposed by a set P of partition vertices $\{v_{\tilde{p}_1}, \ldots, v_{\tilde{p}_{m-1}}\}$ and $\tilde{p}_0 = 0, \tilde{p}_m = N$ which minimises

$$\sum_{a \in A'} l(a) \tag{5.12}$$

such that

$$\sum_{v \in V_k} w(v) \; + \; \sum_{a \in A_k} l(a) \; \leq \; X \tag{5.13}$$

for $k = 1, \ldots, m$?
The sets $A_1, \ldots, A_m$ are defined as

$$A_k = \{(u, v) \in A : \; u \in V_j \; \wedge \; v \in V_k \; \wedge \; j < k\}$$

for $k = 1, \ldots, m$. A' is the union of these

$$A' = \{(u, v) \in A : \; u \in V_i \; \wedge \; v \in V_j \; \wedge \; i \neq j\}$$

Figure 5.7: Definition of SGP

certainly find an optimal partition to be found within the set $S(R) \cup E(R)$ of interval start- and endpoints[6].

2. IP wants to minimise the total number of intervals crossing the partition breakpoints p_k ($k = 1, \ldots, m-1$).

3. Lemma 1 says that the number of intervals in a fragment R_k is the number of intervals that have their startpoint in the partition range $(p_{k-1}, p_k]$ plus the number of intervals that overlap the left border p_{k-1}.

From the first observation we can conclude that we need only the start- and endpoints from a collection R of intervals. We choose the (ordered) set of start- and endpoints as the (ordered) vertex set V of a graph $G = (V, A)$. There are edges only between adjacent vertices. The length of an edge is the number of intervals that include the corresponding points. The optimal solution for SGP will try to minimise the sum of edge lengths that are cut by partition boundaries. This translates into minimising the number of intervals that overlap partition boundaries. This is exactly what is intended by IP (see second observation).

Finally, we assign each vertex v the number of intervals that start at the point that corresponds to v as its weight. The sum of vertex weights in a fragment V_k for SGP is then equivalent to the number of intervals starting at the points that correspond to the vertices in V_k. According to the third observation we need to add the number of intervals that overlap the left border. This number is matched by the lengths of the edge that enters V_k from V_{k-1}. Figure 5.8 summarises a reduction M of an instance $\mathit{ip}(R, X)$ of IP to an instance $\mathit{sgp}(G, X)$ of SGP.

5.6.3 Example

We now show how the instance of IP that was presented in figures 5.2, 5.4 and 5.6 is reduced to an instance of SGP.

We have already noted that not every point of the time range $\{1, 2, \ldots, 20\}$ is a start- or an endpoint for some interval. The graph comprises only 15 timepoints / vertices. By carefully looking at figure 5.2 we can see that there are three intervals starting at timepoint 0, two at point 2, etc. which gives the respective vertex weights $w(0) = 3, w(2) = 2, \ldots$. By definition, there are only edges with relevant lengths between adjacent vertices. We therefore

[6]The reason for including $S(R)$ is convenience. In principle, the reduction that is presented later in this section can be modified to concentrate on the endpoints set $E(R)$ only.

Definition: $M : \mathit{ip}(R, X) \rightarrow \mathit{sgp}(G, X)$

- X remains unchanged for $\mathit{sgp}(G, X)$.
- The graph $G = (V, A)$ is derived in the following way:

$$V \;=\; S(R) \cup E(R) \;=\; \{v_1, \ldots, v_N\}$$

with $v_i < v_{i+1}$ for $i = 1, \ldots, N-1$ and

$$A = \{(v_i, v_{i+1}) : \text{there is an } r \in R \text{ including } v_i \text{ and } v_{i+1}\}$$

- Define the the vertex weights:

$$w(v_i) = \text{number of intervals } \textit{starting} \text{ at } v_i$$

- Define the the edge weights:

$$l(v_i, v_j) = \text{number of intervals including } v_i \text{ and } v_j$$

for $j = i + 1$; for convenience we define $l(v_i, v_j) = 0$ for $j \neq i + 1$.

Figure 5.8: The reduction of an instance of IP to one of SGP.

have to look at adjacent timepoints and count the number of intervals that include these points: there are three intervals including points 0 and 2, five intervals including points 2 and 3, etc. which gives the respective edge lengths $l(0,2) = 3, l(2,3) = 5, \ldots$. Figure 5.9 shows the resulting graph.

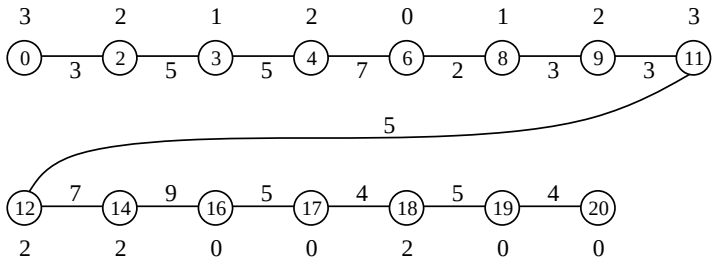


Figure 5.9: Result of reducing the collection of intervals of figures 5.2, 5.4 and 5.6 to a graph.

5.6.4 Correctness

We now prove formally that the reduction that has been presented in the previous section delivers an optimal solution for IP.

Theorem 3 *Using the reduction M described in figure 5.8, each partition P for an instance*

$$\mathit{sgp}(G, X) \;=\; M(\mathit{ip}(R, X))$$

of SGP that satisfies the constraints of SGP satisfies also the constraints of the IP problem for the instance $\mathit{ip}(R, X)$ of IP.

Proof:

The optimal partition P of $\mathit{sgp}(G, X) = M(\mathit{ip}(R, X))$ being also an optimal partition for $\mathit{ip}(R, X)$ means that

$$p_k = v_{\tilde{p}_k} \tag{5.14}$$

for $k = 1, \ldots, m-1$. We show that the constraints imposed by IP and SGP are equivalent, i.e. that (5.12) and (5.13) holds for $\mathit{sgp}(G, X)$ exactly when (5.6) and (5.7) holds for $\mathit{ip}(R, X)$. For this purpose we prove the following relationships:

$$\sum_{p \in P} o_{R}(p) \;=\; \sum_{a \in A'} l(a) \tag{5.15}$$

$$|R_k| \;=\; \sum_{v \in V_k} w(v) \;+\; \sum_{a \in A_k} l(a) \qquad \text{for } k = 1, \ldots, m \tag{5.16}$$

Proof of (5.15)

As a first step we look at how $o_R(p)$ translates into an expression based on the lengths of edges in the graph. Let *next*(p) be the closest start- or endpoint bigger than p. We construct:

$$\begin{aligned} o_R(p) &= |\langle r \in R : r.t_s \leq p < r.t_e \rangle| \\ &= \text{number of intervals that include } p \text{ but do not end at } p \\ &= \text{number of intervals that include } p \text{ and } p+1 \\ &= \text{number of intervals that include } p \text{ and } \textit{next}(p) \\ &= l(v_{\tilde{p}}, \textit{next}(v_{\tilde{p}})) \end{aligned}$$

This means that

$$\sum_{p \in P} o_R(p) = \sum_{p \in P} l(v_{\tilde{p}}, \textit{next}(v_{\tilde{p}})) \qquad (*)$$

We now have to prove that for each $p \in P$ there is a $(v_{\tilde{p}}, \textit{next}(v_{\tilde{p}})) \in A'$ and vice versa, i.e.

$$p \in P \quad \leftrightarrow \quad (v_{\tilde{p}}, \textit{next}(v_{\tilde{p}})) \in A' \qquad (**)$$

But $(**)$ follows from

$$\begin{aligned} A' &= \{(u,v) \in A : u \in V_i \wedge v \in V_j \wedge i \neq j\} \\ &= \{(v_{\tilde{p}}, v_{\tilde{p}+1}) : v_{\tilde{p}} \in P\} \\ &= \{(v_{\tilde{p}}, \textit{next}(v_{\tilde{p}})) : v_{\tilde{p}} \in P\} \\ &= \{(v_{\tilde{p}}, \textit{next}(v_{\tilde{p}})) : p \in P\} \end{aligned}$$

From $(*)$ and $(**)$ follows that (5.15) holds. □

Proof of (5.16)

This proof is based on the third observation made earlier, i.e. lemma 1 with (5.8). It said that the number of intervals in a fragment R_k is (a) the number of intervals having the startpoint in $(p_{k-1}, p_k]$ plus (b) the number of intervals that overlap the left border p_{k-1}. By the definition of $w(v)$ it is obvious that (a) corresponds to

$$\sum_{v \in V_k} w(v) \qquad (*)$$

We now look at the equivalence of (b) to the second sum in (5.13). To this end, we can show that each A_k consists only of one element; remember the

definition of a p_k in (5.14):

$$\begin{aligned} A_k &= \{(u,v) \in A : u \in V_j \wedge v \in V_k \wedge j < k\} \\ &= \{(v_{\tilde{p}_k}, v_{\tilde{p}_k+1})\} \end{aligned}$$

which means that

$$\begin{aligned} \sum_{a \in A_k} l(a) &= l(v_{\tilde{p}_k}, v_{\tilde{p}_k+1}) \\ &= \text{number of intervals including } v_{\tilde{p}_k} \text{ (last vertex in } V_k\text{)} \\ &\quad \text{and } v_{\tilde{p}_k+1} \text{ (first vertex in } V_{k+1}\text{)} \\ &= \text{number of intervals that overlap } p_k \qquad (**) \end{aligned}$$

(5.16) therefore follows from $(*)$ and $(**)$.

□

5.6.5 Optimal Solution for SGP

In this section, we present an algorithm that gives an optimal solution for SGP and – because of the reduction M presented in the previous section – also to IP. The only change to Kernighan's algorithm has to reflect the slightly different calculation of a fragment's weight.

The approach taken by the algorithm `SGP-opt` is similar to the one taken for `IP-opt`. It is based on dynamic programming: the graph is scanned, beginning with v_1 and proceeding one vertex in each step. In step i, i.e. having reached vertex v_i, the algorithm knows an optimal partition for each of the subgraphs containing $v_1, \ldots, v_{i-1}$ respectively. It then seeks the vertex v_j prior to v_i that minimises the partial costs $c(v_i)$ for if the graph ended at v_i and the previous breakpoint had been v_j. The minimising v_j is stored as *pred* (v_i). Finally, when reaching v_n, the optimal partition for the entire graph can be found in $\{$*pred* (v_n), *pred* (*pred* (v_n)), $\ldots\}$.

The following data structures and functions are used by the algorithm in addition to the ones already introduced in the context of SGP:

- For convenience, the algorithm assumes a dummy vertex v_0 with $w(v_0) = 0$.
- The sum of edge lengths that are cut by a partition vertex v_k are stored in

$$\textit{overlaps}\,(v_k) \;=\; \sum_{i \le k < j} l(v_i, v_j) \;=\; \sum_{i=0}^{k} \sum_{j=k+1}^{n} l(v_i, v_j)$$

for $k = 0, \ldots, n$. Please note that

$$\textit{overlaps}\,(v_0) = \textit{overlaps}\,(v_n) = 0$$

and that

$$\sum_{i=0}^{n} \textit{overlaps}\,(v_i) \;=\; \sum_{(v_i,v_j)\in A'} l(v_i, v_j)$$

- The weight of a vertex fragment $\{v_{x+1}, \ldots, v_y\}$ is stored in

$$\begin{aligned} \textit{load}\,(v_x, v_y) &= \sum_{i=x+1}^{y} w(v_i) \;+\; \sum_{i\le x<j\le y} l(v_i, v_j) \\ &= \sum_{i=x+1}^{y} w(v_i) \;+\; \sum_{i=0}^{x}\sum_{j=x+1}^{y} l(v_i, v_j) \end{aligned}$$

  for $0 \le x < y \le n$, i.e. the sum of weights of vertices $v_{x+1}, \ldots, v_y$ plus the sum of lengths of edges starting at or before v_x and ending in some vertex $v_{x+1}, \ldots, v_y$.

- $c(v_i) =$ minimal partial costs for a partition up to vertex v_i.

- *pred* $(v_i) =$ number of vertex preceding v_i that leads to (minimal) partition costs $c(v_i)$.

The algorithm is shown in figure 5.10. It adopts a similar structure to the one used for `IP-opt` in figure 5.5. This is intended to emphasise the similarities between IP and SGP. For a proof of correctness the reader might refer to [Kernighan, 1971].

5.6.6 Example

We want to see how `SGP-opt` works for the graph of figure 5.9 and $X = 10$. Figure 5.11 shows the matrix for the values of *load* (v_i, v_j). This matrix can be pre-computed in $O(n^2)$ steps. It shows the loads of all possible fragments. Because of $X = 10$ we can discard all fragments with a load greater than X, e.g. a fragment $\{v_2, v_3, \ldots, v_7\}$ is not possible because *load* $(v_1, v_7) = 11 \;>\; X$. For graphs resulting from an IP instance, the matrix shows if there exists a partition that satisfies the constraint $|V_k| \le X$ at all: the diagonal shows the loads for the fragments consisting only of one vertex (i.e. one time point). If there was a load greater than X in the diagonal then this would mean that there would be a timepoint that would be included in more than X intervals. Therefore this point could never be part of any partition range because it would cause the

Algorithm SGP-opt

- $c(v_0) = 0$
- **for** $i = 1$ **to** n **do**
  - $J = \{j : 0 \leq j < i \ \wedge \ \mathit{load}\,(v_j, v_i) \leq X\}$
  - **if** $J = \emptyset$ **then**
    output "No optimal partition." ; stop.
  - $c(v_i) = \min_{j \in J}\{\mathit{overlaps}\,(v_j) \ + \ c(v_j)\}$
    Let $\mathit{pred}\,(v_i) = v_j$ for the minimising v_j.
    If there is more than one qualifying v_j then choose the smallest.

  /* output of breakpoints in descending order */
- $p = v_n$
- **while** $p \geq v_1$ **do**
  - $p = \mathit{pred}\,(p)$
  - output p

Figure 5.10: The algorithm SGP-opt for computing an optimal partition for an instance of SGP.

$$\begin{pmatrix}
3 & 5 & 6 & 8 & 8 & 9 & 11 & 14 & 16 & 18 & 18 & 18 & 20 & 20 & 20 \\
 & 5 & 6 & 8 & 8 & 9 & 11 & 14 & 16 & 18 & 18 & 18 & 20 & 20 & 20 \\
 & & 6 & 8 & 8 & 9 & 11 & 14 & 16 & 18 & 18 & 18 & 20 & 20 & 20 \\
 & & & 7 & 7 & 8 & 10 & 13 & 15 & 17 & 17 & 17 & 19 & 19 & 19 \\
 & & & & 7 & 8 & 10 & 13 & 15 & 17 & 17 & 17 & 19 & 19 & 19 \\
 & & & & & 3 & 5 & 8 & 10 & 12 & 12 & 12 & 14 & 14 & 14 \\
 & & & & & & 5 & 8 & 10 & 12 & 12 & 12 & 14 & 14 & 14 \\
 & & & & & & & 6 & 8 & 10 & 10 & 10 & 12 & 12 & 12 \\
 & & & & & & & & 7 & 9 & 9 & 9 & 11 & 11 & 11 \\
 & & & & & & & & & 9 & 9 & 9 & 11 & 11 & 11 \\
 & & & & & & & & & & 9 & 9 & 11 & 11 & 11 \\
 & & & & & & & & & & & 5 & 7 & 7 & 7 \\
 & & & & & & & & & & & & 6 & 6 & 6 \\
 & & & & & & & & & & & & & 5 & 5 \\
 & & & & & & & & & & & & & & 4
\end{pmatrix}$$

Rows: $i = 0, \ldots, 14$
Columns: $j = 1, \ldots, 15$

Figure 5.11: Values for *load* (v_i, v_j) for the graph of figure 5.9.

corresponding fragment to exceed the maximum load X. Thus there would be no partition that satisfied the maximum load constraint. The matrix shows that such situations arise for $X < 9$.

Table 5.2 shows the values calculated by `SGP-opt`. It is similar to table 5.1. It is longer because the graph of figure 5.9 contains not only the endpoints $E(R)$ but also the startpoints $S(R)$. Nevertheless, it delivers the same result as in section 5.5.2.

5.6.7 Run-Time Complexity Analysis

In this section, we analyse the run-time complexity for `SGP-opt`. As we can see from the example in section 5.6.3, the reduction of instances of IP produces a certain type of graph that has only edges between adjacent vertices. We denote such graphs as *IP-graphs*. Actually, this property can be exploited nicely to reduce the run time complexity of `SGP-opt` as we will see by proving the following

Theorem 4 *The run-time complexity of* `SGP-opt` *is* $O(n^3)$ *in the general case and* $O(n^2)$ *for IP-graphs as imposed by the interval partitioning (IP) problem if* n *is the*

i	v_i	J	*overlaps* (v_i)	*pred* (v_i)	$c(v_i)$
1	1	$\{0\}$	3	$v_0 = -1$	0
2	2	$\{0,1\}$	5	$v_0 = -1$	0
3	3	$\{0,1,2\}$	5	$v_0 = -1$	0
4	4	$\{0,1,2,3\}$	7	$v_0 = -1$	0
5	6	$\{0,1,2,3,4\}$	2	$v_0 = -1$	0
6	8	$\{0,1,2,3,4,5\}$	3	$v_0 = -1$	0
7	9	$\{3,4,5,6\}$	3	$v_5 = 6$	2
8	11	$\{5,6,7\}$	5	$v_5 = 6$	2
9	12	$\{5,6,7,8\}$	7	$v_5 = 6$	2
10	14	$\{7,8,9\}$	9	$v_7 = 9$	5
11	16	$\{7,8,9,10\}$	5	$v_7 = 9$	5
12	17	$\{7,8,9,10,11\}$	4	$v_7 = 9$	5
13	18	$\{11,12\}$	5	$v_{12} = 17$	9
14	19	$\{11,12,13\}$	4	$v_{12} = 17$	9
15	20	$\{11,12,13,14\}$	0	$v_{12} = 17$	9

Optimal partition $P = \{v_5, v_7, v_{12}\} = \{6, 9, 17\}$

Table 5.2: Values computed for the graph of figure 5.9 by `SGP-opt` when $X = 10$.

number of vertices in the graph.

Proof

Assume a graph $G = (V, A)$ and let $n = |V|$. The run-time complexity of `SGP-opt` is determined by the following steps:

- computing *overlaps* (v_x) for $x = 1, \ldots, n$,
- computing *load* (v_x, v_y) for $x, y = 1, \ldots, n$,
- stage 1 of the algorithm (initialisation),
- stage 2 (for-loop),
- stage 3 (output)

The complexities of stages 1, 2, 3 do not differ for IP-graphs and the general cases: stage 1 is $O(1)$, stage 2 is $O(n^2)$ because the *min* function is $O(n)$ and stage 3 is $O(n)$.

The generation of *overlaps* (v_x) and *load* (v_x, v_y) depend on the type of the graph:

General case: The general definitions of *overlaps* (v_x) and *load* (v_x, v_y) are as follows:

$$\textit{overlaps}\,(v_x) \;=\; \sum_{i \le x < j} l(v_i, v_j)$$

which can be done by scanning the set A of edges and adding the lengths if the indices i, j satisfy $i \le x < j$. Thus computing *overlaps* (v_x) is $O(|A|)$ and doing it for $x = 1, \ldots, n$ is $O(n \cdot |A|)$ which is at most $O(n^3)$ as $|A| \le n^2$.

$$\textit{load}\,(v_x, v_y) \;=\; \sum_{i=x+1}^{y} w(v_i) \;+\; \sum_{i=0}^{x} \sum_{j=x+1}^{y} l(v_i, v_j)$$

This allows us to derive a recursive equation

$$\textit{load}\,(v_x, v_{y+1}) = \textit{load}\,(v_x, v_y) \;+\; w(v_{y+1}) \;+\; \sum_{i=0}^{x} l(v_i, v_{y+1})$$

which can obviously be computed in $O(n)$ and so it takes $O(n^3)$ time to do it for all $n^2 - \frac{n}{2}$ pairs of x, y with $x < y$.

Thus the general case produces partial complexities $O(n \cdot |A|)$, $O(n^3)$, $O(1)$, $O(n^2)$, $O(n)$ which evolves to $O(n^3)$ in total.

IP-graphs: The IP-graph property of $l(v_i, v_j) = 0$ for all $j \ne i + 1$ allows to compute

$$\textit{overlaps}\,(v_x) \;=\; l(v_x, v_{x+1})$$

in $O(1)$ and in $O(n)$ for all x, and

$$\textit{load}\,(v_x, v_{y+1}) \;=\; \textit{load}\,(v_x, v_y) \;+\; w(v_{y+1})$$

in $O(1)$ and in $O(n^2)$ for all x, y.

Thus the general case produces partial complexities $O(n)$, $O(n^2)$, $O(1)$, $O(n^2)$, $O(n)$ which evolves to $O(n^2)$ in total.

□

Chapter 6

Optimisation of Partitioned Temporal Joins

6.1 Optimisation Process

In this chapter, we build a bridge between

(a) the *analytical part* of this thesis which is formed by chapters 2 to 5 and which introduces, motivates, defines and analyses the problem of processing partitioned temporal joins and

(b) the *synthetical part* which is formed by the following chapters and which is oriented towards a practically applicable and efficient solution for partitioned temporal join processing.

To that end, this chapter summarises the main results of the analysis of part (a) and uses these to design an approach to optimising temporal joins that is based on explicit partitioning. The elaboration of this approach will be presented in the following chapters.

First, we want to focus on the main conclusions that we can draw from what has been discussed so far. In chapter 2, the importance and significance of temporal databases for many applications has been motivated. One obstacle to the incorporation of temporal features into commercial products is the poor performance of operations involving temporal data. One performance critical operator is the temporal join. In chapter 3, we looked at algorithms that are traditionally used for the joins involving an equality join condition. This is the most frequent situation in conventional join processing and most algorithmic techniques have been tuned to perform well in these cases. In this context, explicit partitioning of the data has frequently proved to give the best performance results. In chapter 4, we analysed if and how the techniques that are

used for processing equi-joins can be applied to processing temporal joins. In most cases, this transfer was straightforward. However, techniques that are based on explicit partitioning prove to be tricky: although they can still be expected to be amongst the most efficient, they impose a significant overhead as tuples have to be replicated between the relation fragments. The rate of tuple replication depends (i) on the characteristics of the temporal data and (ii) on the choice of the partition that is used for creating the fragments. While we cannot do anything about (i) we have seen that the choice (ii) of an appropriate partition is a delicate one. In chapter 5, we looked at this choice in more detail and analysed the complexity of the problem of finding an optimal partition. Optimal means that the partition should minimise the total number of tuple replications while creating fragments that do not exceed a certain maximum size. It was shown that this problem has a polynomial solution: there is an algorithm `IP-opt` with a run-time complexity of $O(N^2)$ where N is the number of different start- and endpoints occurring in the relation(s) that are to be partitioned. In practical terms, `IP-opt` is likely to be too inefficient as N is probably huge. From this evolves the need to have heuristic partitioning strategies which are more efficient with respect to the expense of creating only semi-optimal, rather than optimal, partitions.

Such heuristic partitioning strategies form part of a wider optimisation approach. The idea is that a query optimiser can choose the cheapest partition among those produced by various partitioning strategies. In order to determine which partition is the cheapest, we require a cost model of the respective temporal join processing technique. This cost model has to consider

- characteristics of the temporal data (such as values $s_R(t)$ or $o_R(t)$ as used in chapter 5),
- system parameters (such as the number of processing nodes, amount of free memory, current interconnect bandwidth etc.),
- and the respective partition[1] of the time domain.

Figure 6.1 summarises the approach that we propose. It shows the dataflow in the optimisation process for a partitioned temporal join between two temporal relations R and Q[2]. Data is represented as rectangles, computation as ovals.

[1] In the sense as it was defined in chapter 5.

[2] This does not imply a restriction to 2-way temporal joins. The techniques that we propose in this thesis can also be applied to n-way joins with $n \geq 3$.

The entire process consists of four stages corresponding to the four grey boxes in figure 6.1:

1. Firstly, the temporal relations have to be analysed to acquire some information about the structure and characteristics of the temporal data. In its simplest form, this information can be represented by the temporal relations themselves.

   This is not very practical. Alternatively, a data sample can be drawn from the relations. This sample must be big enough to properly represent the characteristics of the data from which it was drawn. The necessary size of a data sample can be determined by the Kolmogorov test statistic [Conover, 1980]. We will return to this issue later. A data sampling approach has been used in the context of band-joins in [DeWitt et al., 1991] and for temporal joins in [Soo et al., 1994].

   A further possibility is to get some meta-information on the data which might be stored in the database catalog. We follow this approach and define *IP-tables* for this purpose. Chapter 7 discusses them in more detail.

2. Based on the information acquired in stage 1 and the systems parameters several strategies can be applied to find suitable partitions for the temporal data. Figure 6.1 assumes that there are three strategies to choose from; in practice there will be more. In chapter 9, we will design and propose several such heuristic partitioning strategies.

3. Using the partitions and information on the temporal data, performance-determining parameters, such as the loads of the fragments R_k and Q_k, can be approximated (e.g. when using data samples) or exactly calculated (e.g. when using complete information on $s_R(t)$, $s_Q(t)$, $o_R(t)$ and $o_Q(t)$). These parameters are then fed into a cost model which derives the processing costs of the partitioned temporal join based on the respective partition and the current system parameters.

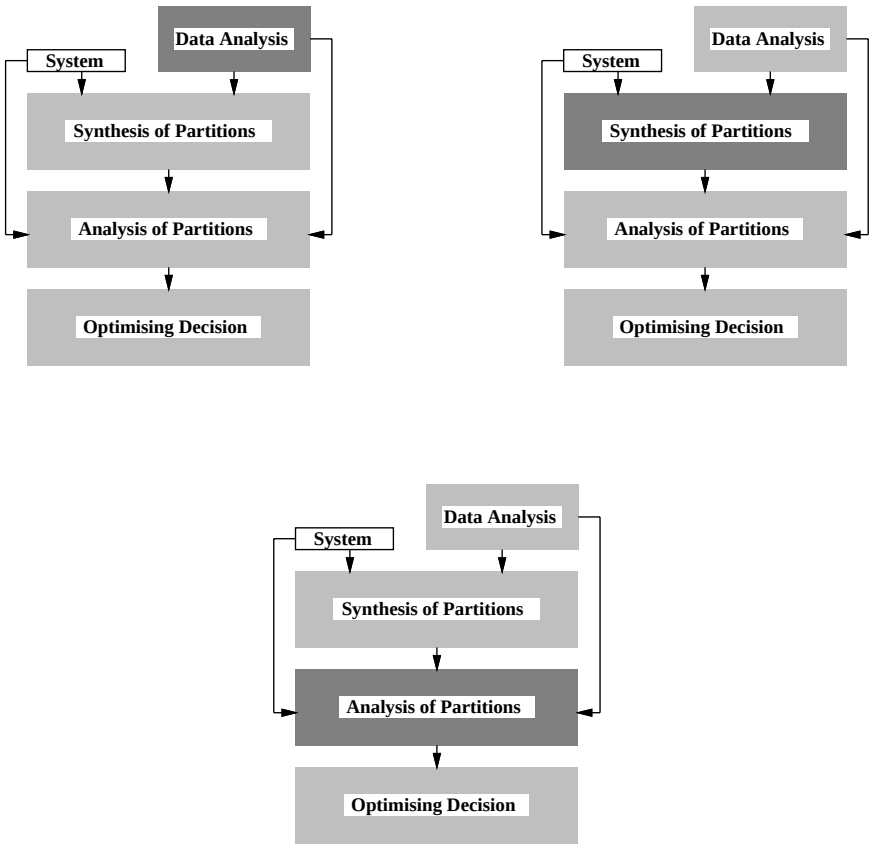
   A crucial part in this stage is the performance model for the respective partitioned temporal join processing technique. In reality, there will be a lot of such techniques and, consequently, the same number of cost models. Frequently, these techniques will be adapted to a target (sequential or parallel) hardware platform. Consequently, there is no single and generally usable cost model for partitioned temporal join processing but several. In chapter 8, we will model the performance of a sequential and a

parallel technique and try to be as general as possible with respect to assumptions about the underlying hardware. These two cost models will also be used for evaluation purposes in chapter 10.

4. Finally, an optimisation decision can be taken and the cheapest partition is chosen.

When looking at the dataflow in figure 6.1 we note that the optimisation process itself is highly parallel: each partitioning strategy initiates an independent thread. Only the final stage is the point of synchronisation when the results of each thread are analysed and the optimisation decision is taken.

In the following chapters, we will use the following simplified version of figure 6.1 to guide you through the optimisation approach by highlighting the stage that is respectively dealt with.



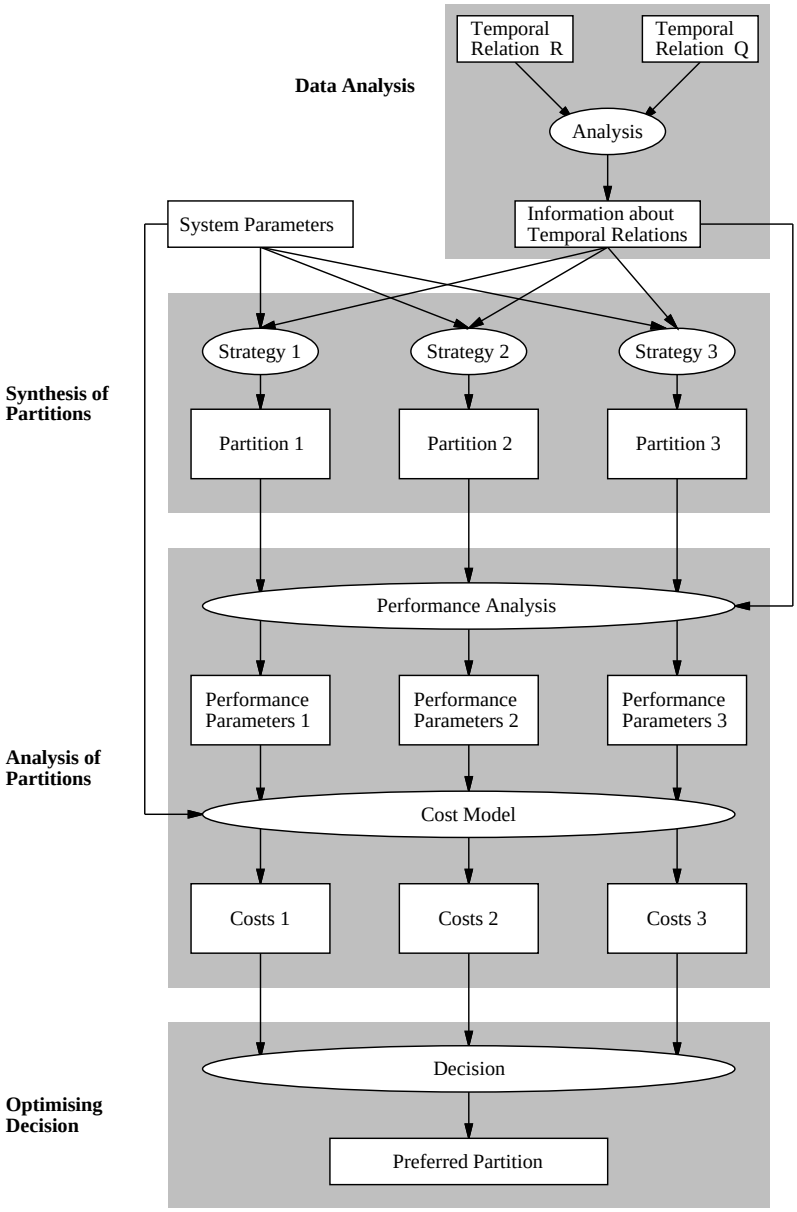


Figure 6.1: Structure of the optimisation process.

6.2 Integration into a Query Optimiser

Let us now consider the question where the optimisation process, as presented in the previous section, would be integrated into a query optimiser. To that end, we have to look at the tasks that are performed by a query optimiser. As a starting point we have to imagine that a query has been translated into an algebraic expression which itself corresponds to a operation tree or query tree. This query tree describes the order in which the individual operations are processed and on which inputs. See figure 6.2 for an example of such a tree. A query optimiser then tries to transform an initial query tree into an equivalent one, i.e. one that yields the same result, but one that implies less costs. There are various possibilities at various levels to do this. Essentially, there are three stages of optimisation [Graefe, 1993]:

1. **Semantic query optimisation**
   At this stage, an optimiser derives, for example, implied predicates using transitivity and other algebraic properties or integrity constraint. From selection conditions, such as $r.A = q.A and q.A = s.A$, it can, for example, derive that $r.A = s.A$ which could possibly help to simplify the original algebraic expression and therefore also the corresponding tree. Another example of semantic query optimisation is to use the implicit fact that an interval's startpoint ts cannot lie beyond its endpoint te, i.e. $t.s \leq t.e$, for simplifying an algebraic expression.

2. **Logical query optimisation**
   At the logical level, the optimiser considers transformations of the query expressions to other, equivalent expressions. The join operation, for example, is commutative and associative [Ryan and Smith, 1995]. Therefore it is

   $$(R \bowtie_C Q) \bowtie_C S \quad = \quad (R \bowtie_C S) \bowtie_C Q \tag{6.1}$$

   Logical query optimisation also considers statistical profiles of the relation, selectivities of selection conditions and estimates sizes of intermediate results from that. In general, it is beneficial to avoid huge intermediate results. This is an important criterion, for example, in order to decide whether the left or the right expression in (6.1) imposes less costs.

3. **Physical query optimisation**
   Finally, at the physical level, an optimiser maps a query tree to the optimal (or at least a near optimal) combination of execution algorithms.

Typically, there is a variety of algorithms for each operation on offer. In order to select the most appropriate algorithm, an optimiser considers whether it can use indices, exploit sort-orders, optimise resource allocation etc. In chapters 3 and 4, we have already noted that the selectivity factor is an important characteristic for deciding on the most appropriate join algorithm. At this stage, the optimiser also employs cost models and performs cost calculations.

We note that these stages might interfere with each other. At the physical level, for example, an optimiser might note that the usage of an index could be exploited if one of the equivalent query expressions was used that have been discarded by the logical optimisation. Thus the physical optimiser might 'ask' to reconsider the expressions in the light of this new information. In fact, the three stages mentioned above have become cumbersome in many modern relational systems.

We now want to look at the integration of the optimisation process of figure 6.1 into an optimiser as it has been described above. From the discussion it becomes obvious that it can form an integral part of the physical optimisation level. The optimisation that we propose can answer two questions in that context:

- Should a temporal join (as it appears in some query) be processed by partitioning over the interval timestamps? Does it achieve less costs than a sort-merge or any other join technique? This could be answered by comparing the cost predictions for partitioned join processing with those of the other techniques. This would require cost models for these other techniques that are based on the same assumptions as those made in chapter 8. It is not our intention to provide these additional cost models in this thesis. But we nevertheless do want to point to this possibility which could be elaborated in future research.

- If the temporal join is to be processed by partitioning over the interval attribute, our optimisation also provides a decision on a suitable, near-optimal partition.

In that sense, our optimisation can form part of the physical optimisation level as it was outlined above. It not only forms part of the optimiser's process of selecting the most appropriate temporal join algorithm but provides also an important input when a partitioning approach has been selected.

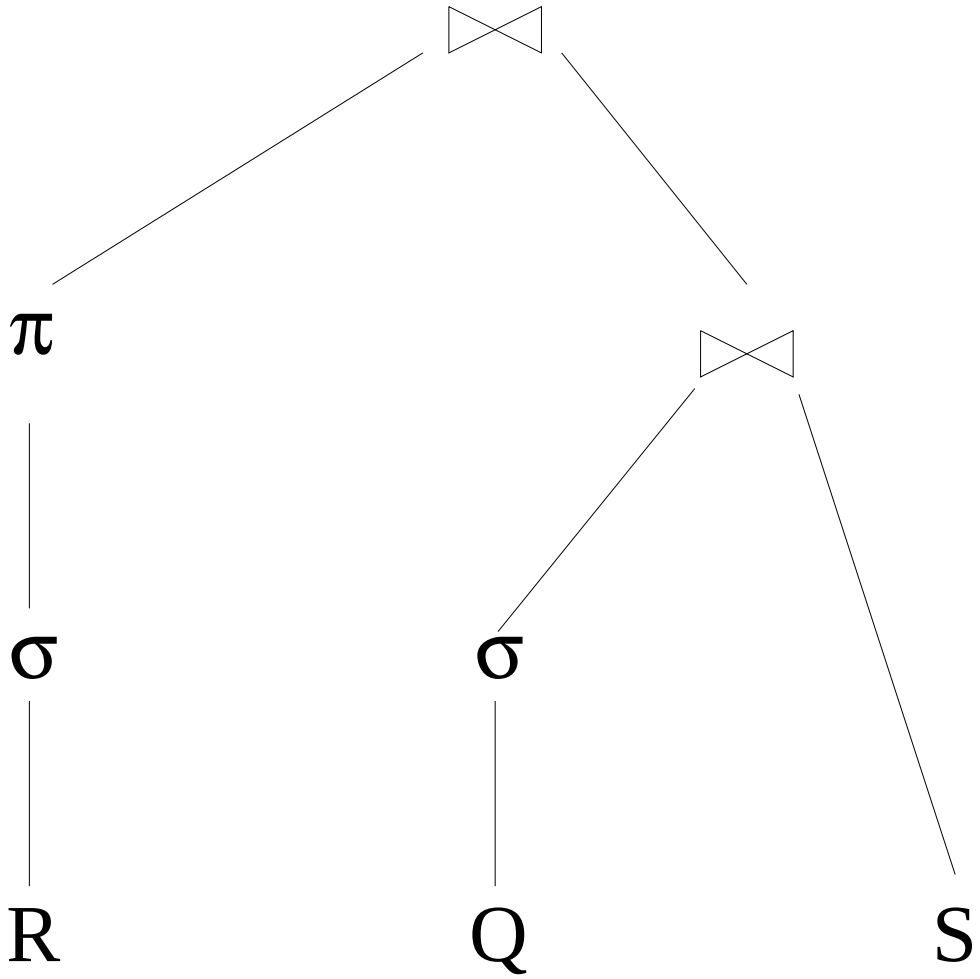
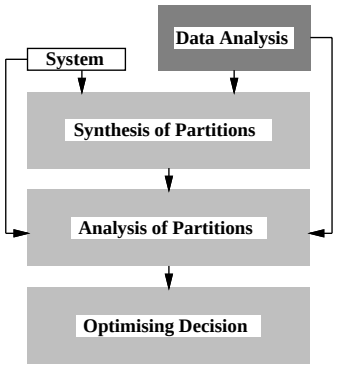


Figure 6.2: A query tree for the relational expression $\pi_A(\sigma_B(R)) \bowtie_C \sigma_D(Q) \bowtie_E S$. The leaves consist of input, internal nodes hold operators.

Chapter 7

IP-Tables

In this chapter, we look at stage 1 of the optimisation process as it has been outlined in the previous chapter: the stage of data analysis.



In general, there are two possibilities for representing the characteristics of the temporal data:

- empirically, e.g. by a data sample, or
- analytically, e.g. by functions s_R, e_R, i_R, o_R, etc.

We opt for the analytical approach because it avoids the fact that implicit information has to be made explicit at one point, thus imposing additional computational effort on later stages of the optimisation process. If the information about the temporal data is to be held explicitly then it has to be computed before it is actually required. Computing figures, such as the $s_R(t)$, $e_R(t)$, ..., from the temporal relation R at optimisation time would be very inefficient. This means that we need a data structure that can store this information efficiently in some convenient place within the database environment, e.g. in the database catalog.

In the following sections, we will motivate and define a data structure that serves for this purpose (sections 7.1 and 7.2). It is called an *IP-table* with *IP* referring to *interval partitioning*. Then we address the question of the size of IP-tables and how it can be reduced if this becomes necessary. This results in two new types of IP-tables (section 7.3). IP-tables, once they are created, can be updated whenever new tuples are inserted into or removed from the corresponding temporal relation. This avoids the necessity to recompute IP-tables after such updates. Section 7.4 looks at this issue. Finally, we present a way in which IP-tables of individual temporal relations can be merged to

derive an IP-table that characterises the collection of intervals that arises from the union of the participating relations (section 7.5). IP-tables can be regarded as a form of histogram for interval data. In section 7.6, we will look at this relationship and try to identify similarities.

7.1 Motivation

Before defining IP-tables in section 7.2 we want to make three observations which serve as a motivation:

- The most important parameters for the cost model of a partitioned join are the cardinalities $|R_k|$ and $|Q_k|$ of the fragments R_k and Q_k (for $k = 1, \ldots, m$). Imagine now a temporal relation R and a partition P with the breakpoints $\{p_1, \ldots, p_{m-1}\}$. Then the number $|R_k|$ of tuples in a fragment R_k can be determined by (i) the number of tuples that overlap from preceding fragments R_j ($j < k$) plus (ii) the tuples that start in the partition range $(p_{k-1}, p_k] = [p_{k-1} + 1, p_k]$ that corresponds to R_k. Obviously (i) corresponds to the number of intervals which contain the first point in the partition range but start before, i.e. the number $o_R(p_{k-1})$, whereas (ii) can be determined by summing up the values $s(p_{k-1} + 1), \ldots, s(p_k)$. Thus

$$|R_k| \quad = \quad o_R(p_{k-1}) \; + \sum_{t=p_{k-1}+1}^{p_k} s_R(t) \tag{7.1}$$

  Several other performance influencing parameters can be calculated in a similar way such as the

$$\textit{total number of overlapping intervals} \quad = \quad \sum_{k=1}^{m-1} o_R(p_k)$$

  or the

$$\begin{aligned} \textit{average interval length} \quad &= \quad \frac{1}{|R|} \cdot \sum_{t=\min T(R)}^{\max T(R)} i_R(t) \\ &= \quad \frac{1}{|R|} \cdot \sum_{t=\min T(R)}^{\max T(R)} s_R(t) + o_R(t-1) \end{aligned}$$

- We need to know only the values of two functions out of s_R, e_R, i_R, o_R for a temporal relation R. The values of the unknown functions can be derived by using equations (5.1), (5.2) and (5.3) or their derivatives in figure 5.1. For the following, we choose s_R and o_R to be stored explicitly whereas e_R and i_R are derived when necessary.

- For our purposes, we require only the values $s_R(t)$ and $o_R(t)$ for those t at which at least one interval starts or ends: for all other timepoints, t, it is $s_R(t) = 0$ and $o_R(t) = o_R(t')$ with t' being the next start- or endpoint (of some interval) before t. This corresponds to the observation made in theorem 1 in section 5.4 and allows us to concentrate on the start- and endpoints[1] of the intervals rather than the entire time span.

7.2 Definition

An IP-table for one or more temporal relations stores information about the temporal structure of the time intervals appearing in these relations. An IP-table is specific to those temporal relations. Figure 7.1 shows the definition of an IP-table for a temporal relation R; the definition for two or more relations works accordingly.

Definition: (complete) IP-table

The *IP-table* for R, $I(R)$, consists of three columns, each with N entries. N is the number of distinct start- and endpoints used in intervals of R:

- The first column contains the values

$$V(R) \;=\; S(R) \cup E(R) \;=\; \{t_1, \ldots, t_N\}$$

  such that[a] $\quad t_{j-1} < t_j \quad$ for $\quad j = 2, \ldots, N$.

- The second column holds the values $s_R(t_j)$ for $j = 1, \ldots, N$.
- The third column holds the values $o_R(t_j)$ for $j = 1, \ldots, N$.

[a] Please remember the comment made in the footnote on page 92 with respect to the notation for conventional and ordered sets.

Figure 7.1: Definition of an IP-table.

We note that an IP-table can be considered as a relation itself. Thus IP-tables which represent a form of metadata are represented in the same logical data model as the data itself. This means that metadata can be accessed in the same way as the data. Many other forms of metadata can also be represented as

[1] As you might remember, theorem 1 says that an optimal partition can be found within the set $E(R)$ of endpoints of intervals in R. To simplify the definition of an IP-table we concentrate on $S(R) \cup E(R)$ at the moment and show a reduction of an IP-table to values $t \in E(R)$ in section 7.3.4.

relations [Date, 1995]. It is a nice side-effect that IP-tables stand in harmony with this generally welcomed feature of relational databases.

In section 7.3, we show how N can be reduced if the IP-table becomes too big. This leads to two variations of the IP-table definition. We will then refer to the original version – as defined in figure 7.1 – as a *complete IP-table.*

As mentioned above, we could alternatively use any pair of the values $s_R(t_j)$, $e_R(t_j)$, $i_R(t_j)$, $o_R(t_j)$ for an IP-table. The missing ones can then be derived by using the equations of figure 5.1. Please note the following: because of the third observation made in section 7.1 it is

$$o_R(t_j - 1) \;=\; o_R(t_j - 2) \;=\; \cdots \;=\; o_R(t_{j-1})$$

for $j = 2, \ldots, N$. Consequently, equation (5.2) can be applied as

$$i_R(t_j) \;=\; s_R(t_j) + o_R(t_{j-1}) \tag{7.2}$$

to the elements $t_j \in V(R)$ for $j = 2, \ldots, N$. This fact also translates into similar changes for the equations of figure 5.1 that were derived from (5.2).

Figure 7.2 shows the example for timestamp intervals of a temporal relation R that has already been used in chapter 5. Intervals are represented as bold bars connecting their start- and endpoint respectively. Figure 7.3 shows the corresponding IP-table $I(R)$ for R (in bold typeface) plus the derivable values $e_R(t_j)$ and $i_R(t_j)$ for demonstration purposes.

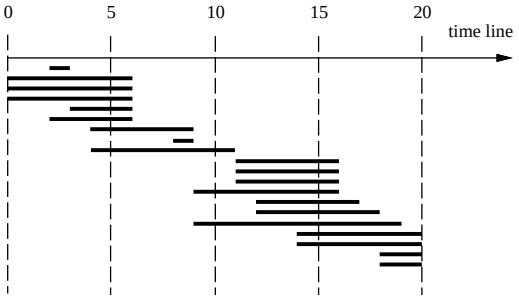


Figure 7.2: An example scenario for timestamp intervals of a temporal relation R.

7.3 Size Considerations

In section 7.2, we assumed that the number N of entries in the IP-table equals the number of distinct interval start- and endpoints, i.e. $N = |V(R)| = |S(R) \cup$

j	t_j	$s_R(t_j)$	$o_R(t_j)$	$e_R(t_j)$	$i_R(t_j)$
1	**0**	**3**	**3**	0	3
2	**2**	**2**	**5**	0	5
3	**3**	**1**	**5**	1	6
4	**4**	**2**	**7**	0	7
5	**6**	**0**	**2**	5	7
6	**8**	**1**	**3**	0	3
7	**9**	**2**	**3**	2	5
8	**11**	**3**	**5**	1	6
9	**12**	**2**	**7**	0	7
10	**14**	**2**	**9**	0	9
11	**16**	**0**	**5**	4	9
12	**17**	**0**	**4**	1	5
13	**18**	**2**	**5**	1	6
14	**19**	**0**	**4**	1	5
15	**20**	**0**	**0**	4	4

Figure 7.3: The IP-table $I(R)$ (in bold typeface) for the intervals in figure 7.2 plus the derivable values $e_R(t_j)$ and $i_R(t_j)$.

$E(R)|$. Critics might argue that – in the worst case – a temporal relation with, for example, one million tuples has a huge IP-table with two million entries and that this might cause the IP-table to be too big to be handled efficiently. In the following, we want to address these concerns and look at sizes of IP-tables in comparison to data samples (section 7.3.1), IP-table sizes for real world temporal relations (section 7.3.2) and two ways for reducing the size of an IP-table in case that it becomes too big (sections 7.3.3 and 7.3.4).

7.3.1 The Size of an IP-Table

In this section, we look at the realistic size of an IP-table and how it compares to sizes of data samples that are used in a typical data sampling approach. The ratio of the sizes of an IP-table $I(R)$ and its corresponding temporal relation R can be calculated by

$$\frac{\textit{size of } I(R)}{\textit{size of } R} = \frac{|V(R)| \cdot \textit{entrysize}}{|R| \cdot \textit{tuplesize}} = \frac{|S(R) \cup E(R)| \cdot \textit{entrysize}}{|R| \cdot \textit{tuplesize}} \tag{7.3}$$

with *entrysize* being the size of an entry in the IP-table and *tuplesize* referring to the size of a tuple of R.

The ratio (7.3) has to be compared to ratios achieved when sampling data.

To that end, we want to look at one such example, i.e. the approach taken in [Soo et al., 1994]. It uses the Kolmogorov test statistic [Conover, 1980] which is frequently employed in data sampling approaches for query optimisation, e.g. in [DeWitt et al., 1991]. The Kolmogorov test is non-parametric which means that it does not make any assumptions about the underlying distributions of the tuples. Soo *et al.* conclude that one has to draw a sample whose size is determined by

$$\textit{sample size in pages} \;=\; \left(1.63 \cdot \frac{\textit{relation size in pages}}{\textit{errorsize}}\right)^2 \qquad (7.4)$$

with *errorsize* being the number of buffer pages that are provided for keeping an overflow of tuples in the buffer. This overflow can be caused by the error difference between the data characteristics of the sample and that of the entire data. Therefore one has to provide a certain buffer space to cope with such an overflow situation. Soo *et al.* optimise *errorsize* in order to minimise the accumulated costs of sampling and joining the two relations. However, the algorithm *determinePartIntervals* provided for that in [Soo et al., 1994] is erroneous as it always reaches the extreme case of drawing the entire relation as a sample. For that reason and in order to get an idea of an actual sample size, we assume the ratio

$$\frac{\textit{relation size in pages}}{\textit{errorsize}}$$

to have a fixed value, for example

- 10:1, which leads to a data sample size of 266 pages according to (7.4), i.e. 3.2% of the original relation which they assume to have 8192 pages, or
- 20:1, which leads to a data sample size of 1063 pages, i.e. 13% of the original relation, or
- 30:1, which leads to a data sample size of 2392 pages, i.e. 29% of the original relation.

Let us now see how these numbers compare to the ratios for IP-tables according to (7.3). Firstly, we determine the size of an IP-table entry, *entrysize*. Such an entry consists of

- a timepoint t_j, which might be represented as 6 bytes[2], and
- the two integers, $s_R(t_j)$ and $o_R(t_j)$, which are usually[3] represented as 4 bytes each.

[2] e.g. one byte per day, month, year, hour, minute, second.
[3] Considering the majority of compilers.

In total, these are 14 bytes. The *tuplesize* can vary widely, depending on the underlying application. Typically, we can assume a *tuplesize* to lie in the range between 100 and 1000 bytes.

The ratio $|V(R)| : |R|$ shows how many new elements are contributed to $V(R)$ on average by a tuple r's interval. A ratio of 0.5 indicates that two intervals contribute one new timepoint to $V(R)$, in the case of 1.0 it is one interval adding one timepoint on average and the worst case is 2.0 with each interval introducing two new timepoints (its start- and endpoint) to the plot. Therefore $|V(R)| : |R|$ is an indicator for observing whether there are many tuples in R that share interval start- and endpoints – in this case the ratio is low – or whether most intervals have start- and endpoints that do not appear in other intervals within R – in this case the ratio is high, reaching 2.0 in the worst case when each interval has a start- and an endpoint that does not appear either as a start- or an endpoint in any other interval. Some applications will impose a low ratio, e.g. in the case of a temporal relation holding air pollution figures that are obtained through periodic measurements. Here, many intervals share the timepoints of the measurements as their start- or endpoints. In other situations, such as a temporal relation storing start- and endtimes of telephone calls or computer accesses, we can expect the start- and endpoints of tuple intervals to be arbitrarily distributed over the timeline, therefore possibly causing a higher ratio than in periodic or other regular applications. In section 7.3.2, examples of various real-world temporal relations are analysed and the respective values of the $|V(R)| : |R|$ ratio are given.

Table 7.1 shows typical values for (7.3) depending on the *tuplesize* and the ratio $|V(R)| : |R|$. For most combinations we get values that are at least as good as those achieved by data samples. But recall that an IP-table provides *precise* information whereas the data sample approach achieves these figures only at the expense of introducing error margins which vary immensely with the sample size.

7.3.2 Realistic Examples

In order to discover realistic values for the $|V(R)| : |R|$ ratio, we analysed four real-world temporal relations:

1. We retrieved accesses to a supercomputer at the Edinburgh Parallel Computing Centre (EPCC). Such login information can be found on the frontends which are machines running the UNIX operating system. On these frontend machines, the `last` command provides access information. Its

tuplesize in bytes	$\|V(R)\| : \|R\|$ 0.25	0.50	0.75	1.00	1.25	1.50	1.75	2.00
100	3.5%	7.0%	10.5%	14.0%	17.5%	21.0%	24.5%	28.0%
200	1.8%	3.5%	5.3%	7.0%	8.8%	10.5%	12.3%	14.0%
300	1.2%	2.3%	3.5%	4.7%	5.8%	7.0%	8.2%	9.3%
400	0.9%	1.8%	2.6%	3.5%	4.4%	5.3%	6.1%	7.0%
500	0.7%	1.4%	2.1%	2.8%	3.5%	4.2%	4.9%	5.6%
600	0.6%	1.2%	1.8%	2.3%	2.9%	3.5%	4.1%	4.7%
700	0.5%	1.0%	1.5%	2.0%	2.5%	3.0%	3.5%	4.0%
800	0.4%	0.9%	1.3%	1.8%	2.2%	2.6%	3.1%	3.5%
900	0.4%	0.8%	1.2%	1.6%	1.9%	2.3%	2.7%	3.1%
1000	0.4%	0.7%	1.1%	1.4%	1.8%	2.1%	2.5%	2.8%

Table 7.1: IP-table sizes as percentages of the original relation.

output typically looks as shown in figure 7.4.
In this example, the main access times are during office hours but possibly also during the weekend or late at night. The dataset comprises 125185 tuples. We refer to it as *EPCC*.

2. Similarly, we looked at a cluster of departmental workstations, mainly used by staff during office hours. The set comprises 27206 tuples. It is referred to as *DEPT*.

3. Next, the logins of a workstation cluster in a student computer laboratory was analysed. Here, the access characteristic is different and mainly influenced by the students' timetables: as an example, one can recognise an accumulation of accesses at times when lectures have just finished. The set comprises 27431 tuples and is referred to as *STUD*.

4. Finally, we analysed the flight schedule to and from Frankfurt Airport. This example differs from the others as departure and arrival times follow certain rules. For example, scheduled times use five-minutes-steps, i.e. there is no departure or arrival time such as 15:03 but times like 15:00, 15:05, 15:10, 15:15 etc. In the case of the computer cluster accesses, times were arbitrary. Here, it is a man-made schedule rather than a random process that generates the temporal data in this case. Figure 7.5 shows an extract of the schedule. For the measurements, the times were converted to Central European Time (CET). The dataset comprises 1995 tuples and is referred to as *FRANKFURT*.

In the first three cases, there are various possibilities to interpret a timestamp: it can be considered as a daily timestamp (ignoring weekday and date information if this is irrelevant); it can also be a timestamp inside a week-long lifespan, thus ignoring the date; we can ignore the month, thus considering the timestamp to define a point with a month-long timestamp. There are more possibilities. We mapped the access data into these three lifespans (day, week, month) with the respective lengths 1440, 10080, 44640 (minutes), thus producing three different temporal relations out of each dataset. The fourth set, *FRANKFURT*, imposed a day-long lifespan of length 1440. In total, we had ten temporal relations for each of which we computed the two values $|V(R)|$ and $|V(R)| : |R|$. The results are shown in table 7.2. The figures prove that for these real-world examples one can expect the corresponding IP-table to be of a reasonable size. The ratios $|V(R)| : |R|$ are far away from the worst case scenario and suggest that IP-table sizes can be expected to correspond to the situations described by the left part of table 7.1.

```
yuh    ftp    alab-16.ed.ac.u  Sun Oct 27 12:03 - 12:03  (00:00)
root   ttyp3  yanis.epcc.ed.a  Sun Oct 27 11:45 - 11:46  (00:00)
yuh    ftp    house.ed.ac.uk   Sun Oct 27 11:36 - 11:36  (00:00)
yuh    ttyp2  alab-16.ed.ac.u  Sun Oct 27 11:32 - 17:05  (05:33)
zxa    ttyp0  bottle.ph.ed.ac  Sun Oct 27 10:42 - 16:07  (05:24)
onb01  ttyp0  aborg.dcs.st-an  Sun Oct 27 08:03 - 08:05  (00:01)
onb01  ttyp0  aborg.dcs.st-an  Sun Oct 27 07:52 - 07:56  (00:03)
yuh    ftp    house.ed.ac.uk   Sat Oct 26 18:47 - 18:48  (00:00)
smith  ttyp1  lilly.glg.ed.ac  Sat Oct 26 18:46 - 11:46  (17:59)
smith  ttyp1  lilly.glg.ed.ac  Sat Oct 26 17:20 - 18:45  (01:24)
```

Figure 7.4: A typical example of login information.

Dataset R	$\|R\|$	Day-Lifepan		Week-Lifespan		Month-Lifespan	
		$\|V(R)\|$	$\|V(R)\| : \|R\|$	$\|V(R)\|$	$\|V(R)\| : \|R\|$	$\|V(R)\|$	$\|V(R)\| : \|R\|$
EPCC	125185	1411	0.01	8286	0.07	29525	0.24
DEPT	27206	1408	0.05	8036	0.30	23877	0.88
STUD	27431	1360	0.05	7228	0.26	21379	0.78
FRANKFURT	1995	288	0.14				

Table 7.2: Characteristics of some real-world temporal relations.

7.3.3 Condensation of IP-Tables

We now present one possibility to reduce the size of an IP-table. The idea is to collapse a certain number of IP-table entries, say a, into one. We call this a

FRA	ACE	HF7667	10:05	14:00
FRA	ACE	DE7966	11:10	14:30
FRA	ACE	DE2662	12:25	15:55
FRA	ACE	DE2662	13:50	17:10
FRA	ACE	DE7512	14:00	17:15
FRA	ADB	LH3806	10:25	14:35
FRA	ADB	TK0904	19:45	23:55
FRA	ADB	AEF8852	21:50	01:50
FRA	ADD	ET0751	00:25	09:50
FRA	ADD	LH0590	09:50	20:10
FRA	ADD	LH0590	10:20	20:55
FRA	ADD	LH0592	10:30	21:15
FRA	ADD	ET0715	21:45	08:05

Figure 7.5: An extract of a flight schedule of Frankfurt Airport.

condensation of an IP-table by a and refer to it as $I'(R, a)$ if the original IP-table is $I(R)$. The parameter a is called the *condensation factor*. Condensation means that N' new timepoints $t'_1, \ldots, t'_{N'}$ are created with

$$N' \;=\; \left\lceil \frac{N}{a} \right\rceil \tag{7.5}$$

For simplicity we assume $N' = N/a$ for a moment. Later we will come back to the situation when this constraint does not hold. The timepoints $\{t_1, t_2, \ldots, t_a\}$ form a new timepoint t'_1, the timepoints $\{t_{a+1}, t_{a+2}, \ldots, t_{2a}\}$ form t'_2 etc. In general, the timepoints $\{t_{(j-1)\cdot a+1}, \ldots, t_{j\cdot a}\}$ form a new timepoint t'_j which gets the value of $t_{j\cdot a}$:

$$t'_j \;=\; t_{j\cdot a} \tag{7.6}$$

with $j = 1, \ldots, N'$. This set of new timepoints is referred to as $V'(R, a)$. See figure 7.6 for a condensation by $a = 2$ for the example of figure 7.2. The definition of the t'_j implies that

$$o_R(t'_j) \;=\; o_R(t_{j\cdot a})$$

This conserves the notion of an overlap: if an interval ends at one of the points within $\{t_{(j-1)\cdot a+1}, \ldots, t_{j\cdot a}\}$ then it logically ends now at t'_j because of the collapse. Therefore it cannot overlap t'_j.

The collapse of $\{t_{(j-1)\cdot a+1}, \ldots, t_{j\cdot a}\}$ also implies that – logically – all intervals that started at one of these points are now considered to start at t'_j. We use a new function s'_R which describes this fact:

$$s'_R(t'_j) \;=\; \sum_{l=(j-1)a+1}^{j\cdot a} s_R(t_l) \tag{7.7}$$

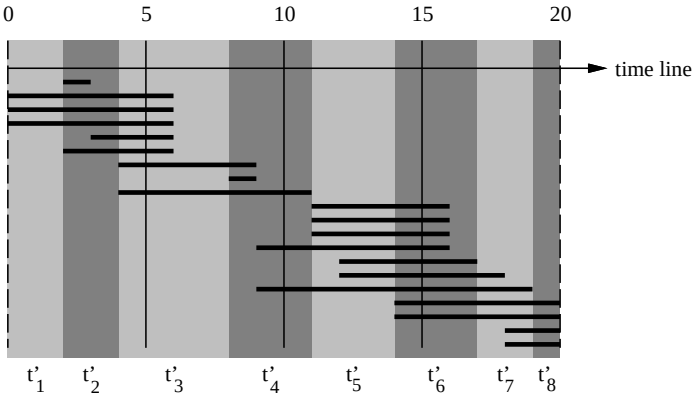


Figure 7.6: Condensation of timepoints with $a = 2$ for the example of figure 7.2.

for all $j = 1, \ldots, N'$ and for all other values $t \notin V'(R, a)$ it is

$$s'_R(t) \;=\; 0$$

Using s'_R instead of s_R in the formula (7.1) still delivers a correct result: if $I'(R, a)$ is used for partitioning rather than $I(R)$ then the resulting partition $P = \{p_1, \ldots, p_{m-1}\}$ is a subset of $V'(R, a)$. We prove this in the following.

Theorem 5 *Let $P = \{p_1, \ldots, p_{m-1}\} \subseteq V'(R, a) \subseteq V(R)$ be a partition for R with $p_{k-1} < p_k$ for $k = 2, \ldots, m-1$, and let $p_0 = t_{\min} - 1$ and $p_m = t'_{N'}$. Then the following holds*

$$\sum_{t=p_{k-1}+1}^{p_k} s_R(t) \;=\; \sum_{t=p_{k-1}+1}^{p_k} s'_R(t) \tag{7.8}$$

for all $k = 1, \ldots, m$.

Proof:

For a $k \in \{1, \ldots, m\}$ let $p_{k-1} = t'_x$ and $p_k = t'_y$ with $x, y \in \{1, \ldots, N'\}$. For convenience, we define $t'_0 = p_0 = t_{\min} - 1$. Then it is

$$\begin{aligned}
\sum_{t=p_{k-1}+1}^{p_k} s_R(t) \;&=\; \sum_{t=t'_x+1}^{t'_y} s_R(t) \\
&=\; \sum_{t=t_{x\cdot a}+1}^{t_{y\cdot a}} s_R(t)
\end{aligned}$$

$$
\begin{aligned}
&= \underbrace{\sum_{t=t_{x\cdot a+1}}^{t_{x\cdot a+1}-1} s_R(t)}_{=0} + \sum_{t=t_{x\cdot a+1}}^{t_{y\cdot a}} s_R(t) \qquad (s_R(t)=0 \quad \text{for} \quad t \notin V(R)) \\
&= \sum_{l=x\cdot a+1}^{y\cdot a} s_R(t_l) \\
&= \underbrace{\sum_{l=x\cdot a+1}^{(x+1)\cdot a} s_R(t_l)}_{s'_R(t'_{x+1})} + \underbrace{\sum_{l=(x+1)\cdot a+1}^{(x+2)\cdot a} s_R(t_l)}_{s'_R(t'_{x+2})} + \cdots + \underbrace{\sum_{l=(y-1)\cdot a+1}^{y\cdot a} s_R(t_l)}_{s'_R(t'_y)} \\
&= \sum_{j=x+1}^{y} s'_R(t'_j) \\
&= \sum_{t=t'_{x+1}}^{t'_y} s'_R(t) \\
&= \underbrace{\sum_{t=t'_x+1}^{t'_{x+1}-1} s'_R(t)}_{=0} + \sum_{t=t'_{x+1}}^{t'_y} s'_R(t) \qquad (s'_R(t)=0 \quad \text{for} \quad t \notin V'(R,a)) \\
&= \sum_{t=t'_x+1}^{t'_y} s'_R(t) \\
&= \sum_{t=p_{k-1}+1}^{p_k} s'_R(t)
\end{aligned}
$$

□

Now we can show the following

Corollary 1 *Let $P = \{p_1, \ldots, p_{m-1}\} \subseteq V'(R) \subseteq V(R)$ be a partition for R with $p_{k-1} < p_k$ for $k = 2, \ldots, m-1$, and let $p_0 = t_{\min} - 1$ and $p_m = t'_{N'}$. Then the following holds*

$$
o_R(p_{k-1}) + \sum_{t=p_{k-1}+1}^{p_k} s_R(t) \;=\; o_R(p_{k-1}) + \sum_{t=p_{k-1}+1}^{p_k} s'_R(t) \tag{7.9}
$$

for all $k = 1, \ldots, m$.

Proof:

Trivial because of theorem 5. □

In the case that $N' \neq (N/a)$ all definitions remain the same as shown above for $j = 1, \ldots, (N' - 1)$. For $j = N'$ we define

$$\begin{aligned} t'_{N'} &= t_N \\ o_R(t'_{N'}) &= o_R(t_N) = 0 \\ s'_R(t'_{N'}) &= \sum_{l=(N'-1)a+1}^{N} s_R(t_l) \end{aligned}$$

for the same reasons as above. However, it pays attention to the fact that $t'_{N'}$ does not comprise the same number a of the original timepoints but only $(N \mod a)$. The proof of theorem 5 is not changed by this situation. Therefore theorem 5 and corollary 1 hold for this case too.

The condensation process divides the size of the IP-table by a. As a consequence, the information in the IP-table becomes coarser and less precise. This might decrease the quality of the resulting partitions but, for example, can make the process of deriving the partition more efficient. Just consider the optimal partitioning algorithm `IP-opt` whose runtime is a function of N. In chapter 10, we will perform experiments with different values of a and look at the impact it has on the quality of the partitions that are derived.

Figure 7.7 shows the condensed IP-table $I'(R, 2)$ for the example of figure 7.2. The notion of condensation can be applied to create a function e'_R with $e'_R(t'_j)$ providing the number of intervals that ended within the condensed timepoint's range, i.e. within $(t'_{j-1}, t'_j]$. Similarly, we can define a function i'_R with $i'_R(t'_j)$ providing the number of intervals that intersect with the range of t'_j, i.e. $(t'_{j-1}, t'_j]$. Then the formulas (5.1), (5.2) and (5.3) can be used by replacing s_R, e_R and i_R through s'_R, e'_R and i'_R respectively. This is straightforward for the same reasons that applied in the case of s_R, e_R and i_R (see section 5.2), just that condensed timepoints are used. Please note that condensation assumes that

$$o_R(t'_j - 1) = o_R(t'_j - 2) = \cdots = o_R(t'_{j-1})$$

for $j = 2, \ldots, N'$. This is another expression of the fact that condensation makes the 'resolution'[4] of the timeline coarser. In summary, the formulas of figure 5.1 apply too as they were derived from (5.1), (5.2) and (5.3). This can be verified for the example of figure 7.7.

[4]By analogy with the sense in which this term is used for images.

j	t'_j	$s'_R(t'_j)$	$o_R(t'_j)$	$e'_R(t'_j)$	$i'_R(t'_j)$
1	**2**	**5**	**5**	0	5
2	**4**	**3**	**7**	1	8
3	**8**	**1**	**3**	5	8
4	**11**	**5**	**5**	3	8
5	**14**	**4**	**9**	0	9
6	**17**	**0**	**4**	5	9
7	**19**	**2**	**4**	2	6
8	**20**	**0**	**0**	4	4

Figure 7.7: The IP-table $I'(R,2)$ (in bold typeface) for the intervals in figure 7.2 plus the values $e'_R(t'_j)$ and $i'_R(t'_j)$.

7.3.4 Endpoint IP-Tables

When looking for an optimal partition in chapter 5, we found out that an optimal partition can always be found within the set $E(R)$ of interval endpoints of a temporal relation R (theorem 1). The proof for this was essentially based on lemma 3. It showed the benefits of using interval endpoints as breakpoints of a partition because this can possibly reduce the number of overlapping intervals. This advantage does not only apply when we look for an optimal partition but shows that interval endpoints are probably good choices for breakpoints of a partition in any case: choosing the breakpoints from the intervals' endpoints should reduce the number of overlapping intervals. Therefore we can reduce an IP-table $I(R)$ to entries concerning endpoints and call this an *endpoint IP-table* $I''(R)$.

Creating an endpoint IP-table is similar to condensing an IP-table as described in the previous section. The only difference is that we collapse those of the original timepoints $t_j \in V(R)$ that are in between two $t_{left}, t_{right} \in E(R)$ with $t_{left} < t_j \leq t_{right}$ into t_{right}. See figure 7.8 for an example of this process. Formally, the creation of an endpoint IP-table can be described like this:

- Let
$$V''(R) \;=\; E(R) \;=\; \{t''_1, \ldots, t''_{N''}\} \;\subseteq\; V(R)$$
with $\quad t''_j < t''_{j+1} \quad$ for $\quad j = 2, \ldots, N''$.
We note that it is always $t_N \in E(R)$. As a consequence we get $t_N = t''_{N''}$.

- Let f be the function that maps the index j of a $t''_j \in V''(R)$ to the index h for a $t_h \in V(R)$ such that $t''_j = t_h = t_{f(j)}$, i.e. $f(j) = h$. To simplify following formulas, we define $f(0) = 0$.

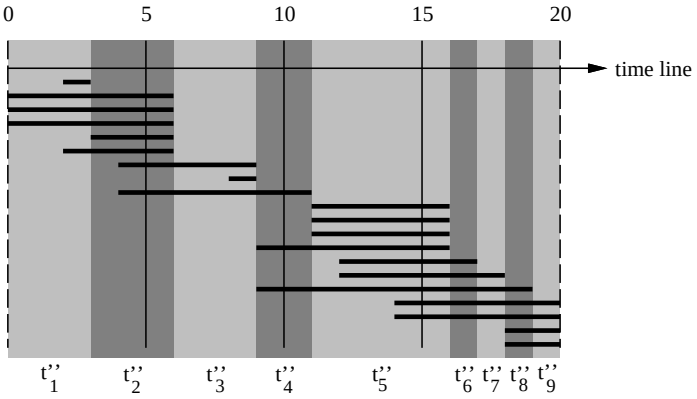


Figure 7.8: Collapsing timepoints into interval endpoints for the example of figure 7.2.

We collapse the original timepoints $\{t_{f(j-1)+1}, \ldots, t_{f(j)}\}$ into the new timepoint t''_j for $j = 1, \ldots, N''$. As in section 7.3.3, this implies

$$o_R(t''_j) \;=\; o_R(t_{f(j)})$$

for all $j = 1, \ldots, N''$.

It also implies that – logically – all intervals that start at one of the original timepoints $\{t_{f(j-1)+1}, \ldots, t_{f(j)}\}$ are now considered to start at t''_j. To reflect this fact, we define a new function s''_R with values

$$s''_R(t''_j) \;=\; \sum_{l=f(j-1)+1}^{f(j)} s_R(t_l) \tag{7.10}$$

for $j = 1, \ldots, N''$ and for all other values $t \notin V''(R)$ it is

$$s''_R(t) \;=\; 0$$

We still get a correct result when Using s''_R instead of s_R in the formula (7.1). This is due to the fact that if $I''(R)$ is used for partitioning rather than $I(R)$ then the resulting partition $P = \{p_1, \ldots, p_{m-1}\}$ is a subset of $V''(R)$, thus $p_k \in V''(R)$ for all $k = 1, \ldots, m-1$. We formally prove this in the following

Theorem 6 *Let $P = \{p_1, \ldots, p_{m-1}\} \subseteq V''(R) \subseteq V(R)$ be a partition for R with $p_{k-1} < p_k$ for $k = 2, \ldots, m-1$, and let $p_0 = t_{\min} - 1$ and $p_m = t''_{N''}$. Then the following holds*

$$\sum_{t=p_{k-1}+1}^{p_k} s_R(t) \;=\; \sum_{t=p_{k-1}+1}^{p_k} s''_R(t) \tag{7.11}$$

for all $k = 1, \ldots, m$.

Proof:

For a $k \in \{1, \ldots, m\}$ let $p_{k-1} = t''_x$ and $p_k = t''_y$ with $x, y \in \{1, \ldots, N''\}$. We define $t''_0 = p_0 = t_{\min} - 1$. Then it is

$$
\begin{aligned}
\sum_{t=p_{k-1}+1}^{p_k} s_R(t) &= \sum_{t=t''_x+1}^{t''_y} s_R(t) \\
&= \sum_{t=t_{f(x)}+1}^{t_{f(y)}} s_R(t) \\
&= \underbrace{\sum_{t=t_{f(x)}+1}^{t_{f(x)+1}-1} s_R(t)}_{=0} + \sum_{t=t_{f(x)+1}}^{t_{f(y)}} s_R(t) \qquad (s_R(t) = 0 \quad \text{for} \quad t \notin V(R)) \\
&= \sum_{h=f(x)+1}^{f(y)} s_R(t_h) \\
&= \underbrace{\sum_{h=f(x)+1}^{f(x+1)} s_R(t_h)}_{s''_R(t''_{x+1})} + \underbrace{\sum_{h=f(x+1)+1}^{f(x+2)} s_R(t_h)}_{s''_R(t''_{x+2})} + \cdots + \underbrace{\sum_{h=f(y-1)+1}^{f(y)} s_R(t_h)}_{s''_R(t''_y)} \\
&= \sum_{j=x+1}^{y} s''_R(t''_j) \\
&= \sum_{t=t''_{x+1}}^{t''_y} s''_R(t) \\
&= \underbrace{\sum_{t=t''_x+1}^{t''_{x+1}-1} s''_R(t)}_{=0} + \sum_{t=t''_{x+1}}^{t''_y} s''_R(t) \qquad (s''_R(t) = 0 \quad \text{for} \quad t \notin V''(R)) \\
&= \sum_{t=t''_x+1}^{t''_y} s''_R(t) \\
&= \sum_{t=p_{k-1}+1}^{p_k} s''_R(t)
\end{aligned}
$$

□

Now we can show the following

Corollary 2 *Let $P = \{p_1, \ldots, p_{m-1}\} \subseteq V''(R) \subseteq V(R)$ be a partition for R with $p_{k-1} < p_k$ for $k = 2, \ldots, m-1$ and let $p_0 = t''_1 - 1$ and $p_m = t''_{N''}$. Then the following*

holds

$$o_R(p_{k-1}) + \sum_{t=p_{k-1}+1}^{p_k} s_R(t) \;=\; o_R(p_{k-1}) + \sum_{t=p_{k-1}+1}^{p_k} s''_R(t) \qquad (7.12)$$

for all $k = 1, \ldots, m$.

Proof:

Trivial because of theorem 6. □

Figure 7.9 shows the endpoint IP-table $I''(R)$ for the example of figure 7.2. In contrast to condensed IP-tables, we do not require the definition of additional functions to make the formulas (5.1), (5.2) and (5.3) work when using s''_R rather than s_R. The reason behind this is that exactly $e_R(t''_j)$ intervals end within the range of t''_j: by definition of an endpoint IP-table there can be no interval ending at a timepoint between t''_{j-1} and t''_j. Therefore the accumulated number of intervals ending within the range of t''_j is $e_R(t''_j)$. For the same reason, $i_R(t''_j)$ can be computed by the number of overlaps occurring at t''_{j-1} plus the number of intervals starting within $(t''_{j-1}, t''_j]$. Thus (5.2) applies when replacing s_R by s''_R. Please note that – as in the case of condensed IP-tables – it is

$$o_R(t''_j - 1) = o_R(t''_j - 2) = \cdots = o_R(t''_{j-1})$$

for $j = 2, \ldots, N''$. (5.1) is not affected as it does not involve s_R.

j	t''_j	$s''_R(t''_j)$	$o_R(t''_j)$	$e_R(t''_j)$	$i_R(t''_j)$
1	**3**	**6**	**5**	1	6
2	**6**	**2**	**2**	5	7
3	**9**	**3**	**3**	2	5
4	**11**	**3**	**5**	1	6
5	**16**	**4**	**5**	4	9
6	**17**	**0**	**4**	1	5
7	**18**	**2**	**5**	1	6
8	**19**	**0**	**4**	1	5
9	**20**	**0**	**0**	4	4

Figure 7.9: The IP-table $I''(R)$ (in bold typeface) for the intervals in figure 7.2 plus the values $e_R(t''_j)$ and $i_R(t''_j)$.

We created the respective endpoint IP-table for the dataset examples that have been described in section 7.3.2. Table 7.3 shows the figures. Having more distributed temporal data, i.e. a longer underlying lifespan of the temporal relation (with a constant number of tuples), causes a greater difference between

the sizes of the endpoint IP-table and a complete IP-table. This means that more space can be saved in these cases. An obvious example can be seen by comparing the figures in the *Month-Lifespan* columns of tables 7.2 and 7.3.

Dataset R	Size $\|R\|$	Day-Lifepan		Week-Lifespan		Month-Lifespan	
		$\|V''(R)\|$	$\|V''(R)\| : \|R\|$	$\|V''(R)\|$	$\|V''(R)\| : \|R\|$	$\|V''(R)\|$	$\|V''(R)\| : \|R\|$
EPCC	125185	1306	0.01	7804	0.06	26370	0.21
DEPT	27206	1313	0.05	7010	0.26	16754	0.62
STUD	27431	1218	0.04	6363	0.23	15731	0.57
FRANKFURT	1995	210	0.11				

Table 7.3: Endpoint IP-table characteristics of some real-world temporal relations.

7.4 Maintaining IP-Tables

The general idea is that there are IP-tables for individual temporal relations stored in the database catalog. Obviously, one wants to avoid that an IP-table $I(R)$ (or $I'(R, a)$ or $I''(R)$) is recomputed each time when a temporal relation R is changed through an update. Essentially, one could take two approaches:

- One can argue that a few new, changed or deleted tuples / intervals within a temporal relation do not translate into severe changes within an IP-table. Consequently, such changes will not have a great impact on the quality of the partitions that are created from the information stored in the IP-table. Only after a certain period, i.e. after a major number of updates have been performed, one should recompute the IP-table. In other words: an IP-table is not updated; only when its information is likely to differ too much from the actual state of the corresponding temporal relation then it is entirely recomputed.
  Obviously, this option is only convenient for situations in which newly inserted data has timestamps that are well distributed over the timeline. In the case of transaction time applications, for example, new data has timestamps beyond the end of the current lifespan. In terms of an IP-table this means that there is one or more 'hot spots' at which values would change. Thus the information provided by an IP-table would soon be obsolete for partitioning purposes if it was not updated.

- A second possibility is to maintain an IP-table. This means that the information within the IP-table is updated each time the corresponding

temporal relation is updated. In this section, we describe the actions that have to be performed on an IP-table when a new tuple is inserted into and when a tuple is removed from the corresponding temporal relation. These actions slightly differ, depending on the type of IP-table that is used: section 7.4.1 describes those for complete IP-tables, section 7.4.2 the ones for condensed IP-tables and section 7.4.3 those for endpoint IP-tables.

7.4.1 Maintaining Complete IP-Tables

An existing IP-table $I(R)$ has to be modified whenever a tuple r is inserted into or deleted from the temporal relation R. If a tuple attribute value is changed then this can be considered as the tuple being removed from R and then being inserted as a new tuple with the changed attribute values. To that end, two algorithms are required, one for each form of update.

Figure 7.10 shows the steps that have to be performed when a tuple r is *inserted* into R. First, it has to be checked whether r's interval start- and endpoints are already in the set $V(R)$. If not then they are added to $V(R)$ respectively. Next, the indices j_s and j_e are set: j_s indicates the $t_{j_s} \in V(R)$ that equals $r.t_s$, j_e does the same for $r.t_e$. Then the value $s[j_s]$ – which stores the value of $s_R(t_{j_s})$ – has to be augmented by 1 as there is now one more interval in R that starts at $t_{j_s} = r.t_s$. Finally, the array $o[j]$ – which stores the values $o_R(t_j)$ – is adapted within a **for**-loop: r overlaps all timepoints t_j with $j_s \leq j < j_e$.

We note that the array notation is used for convenience only. It does not suggest that arrays are the best way to implement the following algorithms. In fact, linked list will probably much more efficient in many respects.

In figure 7.11, we show the modifications that have to be performed when a tuple r is *removed* from R. As in the case of insertion, there are two major stages: the modification of $V(R)$ and the modification of the $s[j]$ and $o[j]$ values. This time, it starts with the latter stage: after determining the indices j_s and j_e as above, $s[j_s]$ is reduced by 1 and so are all $o[j]$ with $j_s \leq j < j_e$. This might lead to the situation that either $r.t_s = t_{j_s}$ or $r.t_e = t_{j_e}$ or both can be removed from $V(R)$ in the case that there are no more intervals in R that start or end at these points. This can be checked by looking at the values of $s_R(t_{j_s})$, $s_R(t_{j_e})$, $e_R(t_{j_s})$, $e_R(t_{j_e})$ of which the first two are explicitly stored as $s[j_s]$ and $s[j_e]$ whereas the latter two can be computed according to the formula given in figure 5.1(b):

$$e_R(t) \;=\; s_R(t) + o_R(t-1) - o_R(t)$$

which translates to

$$e_R(t_j) \;=\; s_R(t_j) + o_R(t_{j-1}) - o_R(t_j)$$

as $o_R(t_j - 1) = o_R(t_j - 2) = \cdots = o_R(t_{j-1})$ for $j = 2, \ldots, N$. If no other intervals start or end at $r.t_s$, i.e. $s_R(r.t_s) = e_R(r.t_s) = 0$ then it is removed from $V(R)$. Similarly, if no other intervals start or end at $r.t_e$, i.e. $s_R(r.t_e) = e_R(r.t_e) = 0$ then it is removed from $V(R)$.

```
/* Adapt I(R) when a tuple r with [r.t_s, r.t_e] is inserted into R */

/* V(R) = {t_1, ..., t_N} with t_{j-1} < t_j for j = 2, ..., N */
/* values for s_R(t_j) are in s[j] with t_j ∈ V(R) */
/* values for o_R(t_j) are in o[j] with t_j ∈ V(R) */

if r.t_s ∉ V(R) then
    V(R) = V(R) ∪ {r.t_s}
    initialise a value for r.t_s in s[ ] with 0
    initialise a value for r.t_s in o[ ] with o_R(max{x ∈ V(R) : x < r.t_s})
fi
if r.t_e ∉ V(R) then
    V(R) = V(R) ∪ {r.t_e}
    initialise a value for r.t_e in s[ ] with 0
    initialise a value for r.t_e in o[ ] with o_R(max{x ∈ V(R) : x < r.t_e})
fi

/* Determine the indices of r.t_s and r.t_e within V(R) */
/* V(R) = {t_1, ..., t_N} with t_{j-1} < t_j for j = 2, ..., N */
j_s = the j ∈ {1, ..., N} such that r.t_s = t_j ∈ V(R)
j_e = the j ∈ {1, ..., N} such that r.t_e = t_j ∈ V(R)

/* Update s[ ] and o[ ] */

s[j_s] = s[j_s] + 1

for j = j_s to (j_e - 1) do
    o[j] = o[j] + 1
od
```

Figure 7.10: The insertion algorithm for complete IP-tables.

7.4.2 Maintaining Condensed IP-Tables

The insertion and deletion algorithms described in section 7.4.1 have to be modified in the case of a condensed IP-table $I'(R, a)$. They have to incor-

```
/* Adapt I(R) when a tuple r with [r.t_s, r.t_e] removed from R */

/* Determine the indices of r.t_s and r.t_e within V(R) */
/* V(R) = {t_1, ..., t_N} with t_{j-1} < t_j for j = 2, ..., N */
j_s  = the j ∈ {1, ..., N} such that r.t_s = t_j ∈ V(R)
j_e  = the j ∈ {1, ..., N} such that r.t_e = t_j ∈ V(R)

/* values for s_R(t_j) are in s[j] for j = 1, ..., N */
s[j_s] = s[j_s] − 1

/* values for o_R(t_j) are in o[j] for j = 1, ..., N */
for j = j_s to (j_e − 1) do
    o[j] = o[j] − 1
od

/* Remove r.t_s if there are no other intervals starting or ending at r.t_s */
/* e = number of intervals ending at r.t_s */
/* Assume t_0 = −∞ and s[0] = o[0] = 0. */
e = s[j_s] + o[j_s − 1] − o[j_s]
if s[j_s] = 0 and e = 0 then
    V(R) = V(R) − {r.t_s}
    remove s[j_s] from s[ ]
    remove o[j_s] from o[ ]
fi

/* Remove r.t_e if there are no other intervals starting or ending at r.t_e */
/* e = number of intervals ending at r.t_e */
e = s[j_e] + o[j_e − 1] − o[j_e]
if j_s ≠ j_e and s[j_e] = 0 and e = 0 then
    V(R) = V(R) − {r.t_e}
    remove s[j_e] from s[ ]
    remove o[j_e] from o[ ]
fi
```

Figure 7.11: The deletion algorithm for complete IP-tables.

porate the notion of timepoints having been collapsed into one timepoint, i.e. that an interval $[r.t_s, r.t_e]$ might not have its start- and endpoints within the set $V'(R,a)$. Therefore, we have to determine the timepoints t'_{j_s} and t'_{j_e} of $V'(R,a)$ which represent $r.t_s$ and $r.t_e$ respectively. In the case of r being inserted into R, one has to include $r.t_e$ if r's timestamp falls partly or entirely beyond the current value of $t'_{N'}$. Such a situation is characterised by $r.t_e > t'_{N'}$. Similarly, $r.t_e$ can be removed on deletion of r if there are no more intervals ending at $t'_{N'}$, i.e. if all intervals have ended before, at $t'_{N'-1}$. At the opposite end, we can remove t'_1 if there are no more intervals starting at t'_1. Apart from these modifications, the insertion and deletion algorithms remain the same. They are shown in figures 7.12 and 7.13.

From these algorithms it is apparent that condensed IP-tables can not be maintained without a loss of accuracy. Basically, once the condensation of timepoints has been performed, one cannot control that a condensed timepoint t'_j still represents a original timepoints of $V(R)$. After several insertions or deletions this number might have changed. The only control that can be performed is the one over t'_1 and $t'_{N'}$ which can be removed in case that they become obsolete. Therefore one can expect that the quality of information provided by a condensed IP-table decreases with an increasing amount of updates. This suggests that condensed IP-table might need to be recomputed periodically, in particular if insertions or deletions concentrate on specific parts of the timeline.

7.4.3 Maintaining Endpoint IP-Tables

Similar to the case of condensed IP-tables, the insertion and deletion algorithms have to be changed when using endpoint IP-tables. However, it is possible to accurately maintain the set $V''(R)$ of timepoints within an IP-table.

Figure 7.14 shows the actions that have to be performed when a tuple r is inserted into the temporal relation R. If r's endpoint $r.t_e$ is not contained in $V''(R)$ then it is added. Consequently, there is always a $t''_{j_e} \in V''(R)$ such that $t''_{j_e} = r.t_e$ when it comes to the stage of modifying the $s[j]$ and $o[j]$ values. In contrast to j_e, the index j_s is determined as in the case of a condensed IP-table by looking for the nearest $t''_{j_s} \in V''(R)$ such that $r.t_s \leq t''_{j_s}$. The modification of the $s[\,]$ and $o[\,]$ arrays works as for complete and condensed IP-tables.

The deletion algorithm for endpoint IP-tables is straightforward. It is shown in figure 7.15. It determines j_s and j_e as in the case of insertion, then modifies the $s[\,]$ and $o[\,]$ arrays before finally checking whether the $r.t_e = t''_{j_e}$ is the endpoint of an interval other than r. If it is not then it can be removed from the

/* Adapt $I'(R,a)$ when a tuple r with $[r.t_s, r.t_e]$ is inserted into R */

/* $V'(R,a) = \{t'_1, \ldots, t'_{N'}\}$ with $t'_{j-1} < t'_j$ for $j = 2, \ldots, N'$ */
/* values for $s'_R(t'_j)$ are in $s[j]$ with $t'_j \in V'(R,a)$ */
/* values for $o_R(t'_j)$ are in $o[j]$ with $t'_j \in V'(R,a)$ */
if $r.t_e > t'_{N'}$ **then**
    $V'(R,a) = V'(R,a) \cup \{r.t_e\}$
    initialise a value for $r.t_e$ in $s[\,]$ with 0
    initialise a value for $r.t_e$ in $o[\,]$ with $o_R(\max\{x \in V'(R,a) : x < r.t_e\})$
fi

/* Determine the indices of the condensed timepoints to */
/* which $r.t_s$ and $r.t_e$ belong */
/* Assume a $t'_0 = -\infty$ */
j_s = the $j \in \{1, \ldots, N'\}$ such that $t'_{j-1} < r.t_s \leq t'_j \in V'(R,a)$
j_e = the $j \in \{1, \ldots, N'\}$ such that $t'_{j-1} < r.t_e \leq t'_j \in V'(R,a)$

/* Update $s[\,]$ and $o[\,]$ */

$s[j_s] = s[j_s] + 1$

for $j = j_s$ **to** $(j_e - 1)$ **do**
    $o[j] = o[j] + 1$
od

Figure 7.12: The insertion algorithm for condensed IP-tables.

/* Adapt $I'(R,a)$ when a tuple r with $[r.t_s, r.t_e]$ removed from R */

/* Determine the indices of the condensed timepoints to */
/* which $r.t_s$ and $r.t_e$ belong */
/* Assume a $t'_0 = -\infty$ */
j_s = the $j \in \{1,\ldots,N'\}$ such that $t'_{j-1} < r.t_s \leq t'_j \in V'(R,a)$
j_e = the $j \in \{1,\ldots,N'\}$ such that $t'_{j-1} < r.t_e \leq t'_j \in V'(R,a)$

/* values for $s'_R(t'_j)$ are in $s[j]$ for $j = 1,\ldots,N'$ */
$s[j_s] = s[j_s] - 1$

/* values for $o_R(t'_j)$ are in $o[j]$ for $j = 1,\ldots,N'$ */
for $j = j_s$ **to** $(j_e - 1)$ **do**
    $o[j] = o[j] - 1$
od

/* Remove t'_1 if there are no more intervals starting at t'_1 */
if $s[1] = 0$ **then**
    $V'(R,a) = V'(R,a) - \{t'_1\}$
    remove $s[1]$ from $s[\,]$
    remove $o[1]$ from $o[\,]$
fi

/* Remove $t'_{N'}$ if there are no more intervals ending at $t'_{N'}$ */
/* e = number of intervals ending at $t'_{N'}$ */
$e = s[N'] + o[N'-1] - o[N']$
if $e = 0$ **then**
    $V(R) = V(R) - \{t'_{N'}\}$
    remove $s[N']$ from $s[\,]$
    remove $o[N']$ from $o[\,]$
fi

Figure 7.13: The deletion algorithm for condensed IP-tables.

set $V''(R)$. This removal is not trivial as the value of $s[j_e]$ has to be incorporated into the one of the timepoint t''_{j_e+1} which follows t''_{j_e} within the ordered set $V''(R)$.

/* Adapt $I''(R)$ when a tuple r with $[r.t_s, r.t_e]$ is inserted into R */

/* $V''(R) = \{t''_1, \ldots, t''_{N''}\}$ with $t''_{j-1} < t''_j$ for $j = 2, \ldots, N''$ */
/* values for $s''_R(t''_j)$ are in $s[j]$ with $t''_j \in V''(R)$ */
/* values for $o_R(t''_j)$ are in $o[j]$ with $t''_j \in V''(R)$ */
if $r.t_e \notin V''(R)$ **then**
    $V''(R) = V''(R) \cup \{r.t_e\}$
    initialise a value for $r.t_e$ in $s[\,]$ with 0
    initialise a value for $r.t_e$ in $o[\,]$ with $o_R(\max\{x \in V''(R) : x < r.t_e\})$
fi

/* Determine the indices of the endpoints to which */
/* $r.t_s$ and $r.t_e$ belong */
/* Assume a $t''_0 = -\infty$ */
j_s = the $j \in \{1, \ldots, N''\}$ such that $t''_{j-1} < r.t_s \leq t''_j \in V''(R)$
j_e = the $j \in \{1, \ldots, N''\}$ such that $r.t_e = t''_j \in V''(R)$

/* Update $s[\,]$ and $o[\,]$ */

$s[j_s] = s[j_s] + 1$

for $j = j_s$ **to** $(j_e - 1)$ **do**
    $o[j] = o[j] + 1$
od

Figure 7.14: The insertion algorithm for endpoint IP-tables.

```
/* Adapt I''(R) when a tuple r with [r.t_s, r.t_e] removed from R */

/* Determine the indices of the endpoints to which */
/* r.t_s and r.t_e belong */
/* Assume a t''_0 = -∞ */
j_s  = the j ∈ {1, ..., N''} such that t''_{j-1} < r.t_s ≤ t''_j ∈ V''(R)
j_e  = the j ∈ {1, ..., N''} such that r.t_e = t''_j ∈ V''(R)

/* values for s''_R(t''_j) are in s[j] for j = 1, ..., N'' */
s[j_s] = s[j_s] - 1

/* values for o_R(t''_j) are in o[j] for j = 1, ..., N'' */
for j = j_s to (j_e - 1) do
    o[j] = o[j] - 1
od

/* Remove r.t_e if there are no more intervals ending at r.t_e */
/* e = number of intervals ending at r.t_e */
e = s[j_e] + o[j_e - 1] - o[j_e]
if e = 0 then
    if j_e < N'' then
        s[j_e + 1] = s[j_e + 1] + s[j_e]
    fi
    V(R) = V(R) - {r.t_e}
    remove s[j_e] from s[]
    remove o[j_e] from o[]
fi
```

Figure 7.15: The delete algorithm for endpoint IP-tables.

7.5 Merging IP-Tables

Two (or more) IP-tables of two (or more) temporal relations can be merged into one IP-table that describes the timestamp characteristics of the union of these relations. This is very useful as we can precompute the IP-tables for individual relations and merge them when optimising a temporal join between those relations. This is only relevant for join algorithms that require two or more input relations to be partitioned along certain constraints. Partitioning then needs information on all these relations, i.e. we need the IP-table of the union of the relations. This leads to the layout of the data-analysis stage within the optimisation process that is shown in figure 7.16.

However, merging is not only relevant in the context of optimisation although the latter is the main purpose for which we will use it. It can also be considered as a general technique for updating individual IP-tables: assume a data warehouse that is updated over night by inserting a batch of new data that has been accumulated during the day. One could then simply create a temporary IP-table for this batch and merge it with the existing IP-table in order to get an updated IP-table.

The different types of IP-tables require slightly different algorithms for merging them. Condensed and endpoint IP-tables can be treated equally due to the analogy in collapsing timepoints. This led to analogous definitions of the s'_R and s''_R functions. To stress this analogy, we will refer to condensed and endpoint IP-tables as *incomplete IP-tables* in the remainder of this section.

In the following, we give the algorithms for the case that two IP-tables are to be merged. A three- or more-way merge can easily be derived from that, similarly to the numerous algorithms that are based on merging several streams of data, such as sort-merge joins (see chapters 3 and 4 for example) or the merge-sort algorithm [Knuth, 1973]. For the two-way merge we have to consider three cases:

- Two complete IP-tables are merged. This case is discussed in section 7.5.1.
- Two incomplete IP-tables are merged. This is discussed in section 7.5.2.
- A complete IP-table is merged with an incomplete IP-table. This case is discussed in section 7.5.3.

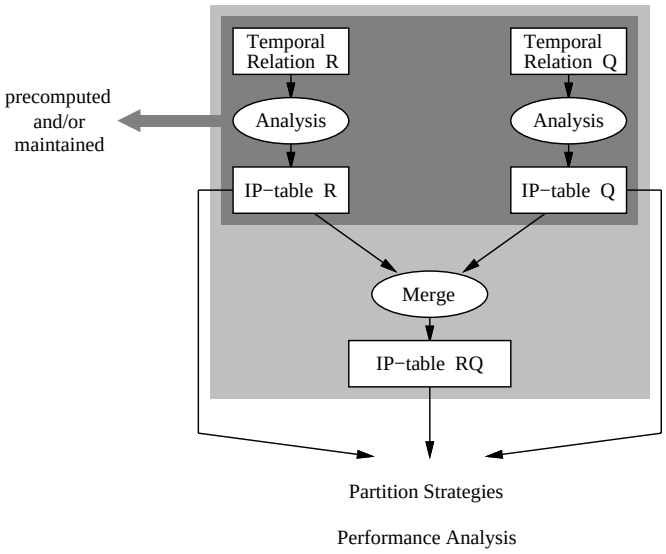


Figure 7.16: Acquiring information about (temporal) characteristics of temporal relations by using IP-tables

7.5.1 Merging Complete IP-Tables

The basic merging process is fairly straightforward: Imagine two complete IP-tables $I(R)$ and $I(Q)$ of two relations R and Q participating in a temporal join $R \bowtie Q$. They have timepoint sets $V(R)$ and $V(Q)$ and functions s_R, o_R and s_Q, o_Q respectively. These two tables can be merged into one IP-table $I(R \cup Q)$ with timepoint set

$$V(R \cup Q) \;=\; V(R) \cup V(Q)$$

and functions $s_{R\cup Q}$, $o_{R\cup Q}$ defined as

$$s_{R\cup Q}(t) \;=\; s_R(t) + s_Q(t) \tag{7.13}$$

$$o_{R\cup Q}(t) \;=\; o_R(t) + o_Q(t) \tag{7.14}$$

We note that the IP-table of R might not hold values for all $t \in V(Q)$ and, similarly, the IP-table of Q might not for all $t \in V(R)$. This is not actually a problem as the missing values can be derived by using the third observation made in section 7.1: it is

$$s_R(t) \;=\; 0 \tag{7.15}$$

$$o_R(t) \;=\; o_R(\min\{x \in V(R) : x \geq t\}) \tag{7.16}$$

for all $t \notin V(R)$ and in particular for those $t \in V(Q) - V(R)$ in the case that the IP-tables are merged. The same applies vice versa when values for s_Q and o_Q have to be derived.

The correctness of (7.13) is trivial: if $s_R(t)$ intervals in R start at time t and $s_Q(t)$ intervals in Q start at t then there are $s_R(t) + s_Q(t)$ intervals starting at t in $R \cup Q$. Similarly for (7.14): if $o_R(t)$ intervals in R overlap timepoint t and $o_Q(t)$ intervals in Q do the same then there are $o_R(t) + o_Q(t)$ overlapping t in $R \cup Q$.

Figure 7.17 shows the algorithm that merges two complete IP-tables $I(R)$ and $I(Q)$ into one IP-table $I(R \cup Q)$ that describes the characteristics of the intervals in $R \cup Q$. The timepoint sets

$$\begin{aligned} V(R) &= \{x_1, \ldots, x_A\} \\ V(Q) &= \{y_1, \ldots, y_B\} \end{aligned}$$

are merged into the set

$$V(R \cup Q) \;=\; V(R) \cup V(Q) \;=\; \{t_1, \ldots, t_N\}$$

The values $s_{R\cup Q}(t_j)$, $s_R(x_l)$, $s_Q(y_h)$ are stored in arrays $s_{R\cup Q}[\,]$, $s_R[\,]$, $s_Q[\,]$ respectively ($j = 1, \ldots, N$; $l = 1, \ldots, A$; $h = 1, \ldots, B$). Similarly, $o_{R\cup Q}(t_j)$, $o_R(x_l)$, $o_Q(y_h)$ are respectively stored in the arrays $o_{R\cup Q}[\,]$, $o_R[\,]$, $o_Q[\,]$. The algorithm mainly consists of a **while**-loop that merges the timepoints of $V(R)$ and $V(Q)$ and calculates their values $s_{R\cup Q}[j]$ and $o_{R\cup Q}[j]$ according to (7.13) and (7.14). This **while**-loop stops when all of the timepoints of at least one of these sets has been merged into $V(R \cup Q)$. Then there might be timepoints that have not been processed yet. This is done by one of the following two **while**-loops. Finally, the cardinality N of $V(R \cup Q)$ is set.

7.5.2 Merging Incomplete IP-Tables

Merging two incomplete IP-tables is similar to merging two complete IP-tables: equations (7.14) and (7.16) still apply when using timepoint sets like $V'(R, a)$, $V'(Q, b)$, $V''(R)$ or $V''(Q)$. The difference lies in the different properties of functions like s_R and its respective counterparts s'_R or s''_R[5]: equation (7.13) does not provide the correct result when s-labelled functions are replaced by their s'-labelled counterparts. Therefore we require another sensible way to calculate the values of $s'_{R\cup Q}$.

Let us assume that $V'(R, a) = \{x_1, \ldots, x_A\}$ with $x_{l-1} < x_l$ for $l = 2, \ldots, A$, and similarly that $V'(Q, b) = \{y_1, \ldots, y_B\}$ with $y_{h-1} < y_h$ for $h = 2, \ldots, B$. The notion behind the definition of $s'_R(x_l)$ was that there are $s'_R(x_j)$ intervals

[5]In the remainder of this subsection we will only use the notation for condensed IP-tables. This is for improving the readability of the text. Nevertheless, everything that applies to condensed IP-tables equally applies to endpoint IP-tables in this context.

```
/* Merge two complete IP-tables I(R) and I(Q) */
/* V(R) = {x_1, ..., x_A} with x_{l-1} < x_l for l = 2, ..., A */
/* V(Q) = {y_1, ..., y_B} with y_{h-1} < y_h for h = 2, ..., B */

V(R ∪ Q) = ∅
l        = 1    /* index of next element of V(R) to be merged */
h        = 1    /* index of next element of V(Q) to be merged */
j        = 1    /* index of next element of V(R ∪ Q) to be created */

while l ≤ A and h ≤ B do      /* merge tables */
    o_{R∪Q}[j] = o_R[l] + o_Q[h]
    if x_l = y_h then
      s_{R∪Q}[j]   = s_R[l] + s_Q[h]
      V(R ∪ Q      = V(R ∪ Q) ∪ {x_l}
      l            = l + 1
      h            = h + 1
    else if x_l < y_h then
      s_{R∪Q}[j]   = s_R[l]
      V(R ∪ Q      = V(R ∪ Q) ∪ {x_l}
      l            = l + 1
    else    /* x_l > y_h */
      s_{R∪Q}[j]   = s_Q[h]
      V(R ∪ Q      = V(R ∪ Q) ∪ {y_h}
      h            = h + 1
    fi
    j = j + 1
od

if h > B then
  while l ≤ A do          /* add rest of V(R) */
      s_{R∪Q}[j]   = s_R[l]
      o_{R∪Q}[j]   = o_R[l]
      V(R ∪ Q      = V(R ∪ Q) ∪ {x_l}
      l            = l + 1
      j            = j + 1
  od
else
  while h ≤ B do          /* add rest of V(Q) */
      s_{R∪Q}[j]   = s_Q[h]
      o_{R∪Q}[j]   = o_Q[h]
      V(R ∪ Q      = V(R ∪ Q) ∪ {y_h}
      h            = h + 1
      j            = j + 1
  od
fi

N = j - 1
```

Figure 7.17: The merge algorithm for two complete IP-tables.

starting within the time range $(x_{l-1}, x_l]$. If we make the assumption that these intervals' startpoints' distribution is uniform then there are

$$\frac{1}{x_l - x_{l-1}} \cdot s'_R(x_l)$$

intervals in R starting at any point $t \in (x_{l-1}, x_l]$. Thus there are

$$\frac{z}{x_l - x_{l-1}} \cdot s'_R(x_l)$$

intervals starting in some range within $(x_{l-1}, x_l]$ that comprises z timepoints. In particular, this applies to a range $(t_{j-1}, t_j]$ with $x_{l-1} \leq t_{j-1} < t_j \leq x_l$ as being used in the merging process with t_{j-1} and t_j being elements of a merged timepoint set $V(R \cup Q)$: there are

$$\frac{t_j - t_{j-1}}{x_l - x_{l-1}} \cdot s'_R(x_l)$$

intervals starting in $(t_{j-1}, t_j]$. As the quotient might lead to a non-integer result we have to round the result of (7.17) to get an integer:

$$\mathit{round}\left(\frac{t_j - t_{j-1}}{x_l - x_{l-1}} \cdot s'_R(x_l)\right) \tag{7.17}$$

Similarly, we can derive values for the intervals in Q.

The significance of (7.17) is that it allows us to provide an approximation for a $s'_R(t_j)$ with $s'_R(t_j)$ providing the number of intervals that have started since $t_{j-1} < t_j$. The novelty is that this is possible for any pair $\{t_{j-1}, t_j\} \subseteq L(R \cup Q)$ with $x_{l-1} \leq t_{j-1} < t_j \leq x_l$ for some $l \in \{1, \ldots, A\}$. Consequently, we can calculate a value $s'_{R \cup Q}(t)$ in the following way: we assume that the merging process has reached a stage such

- that the elements

  $$\begin{aligned} \{x_1, \ldots, x_{l-1}\} &\subset V'(R, a) \\ \{y_1, \ldots, y_{h-1}\} &\subset V'(Q, b) \end{aligned}$$

  have been processed for some $l \in \{1, \ldots, A\}$ and some $h \in \{1, \ldots, B\}$,

- that there are dummy values

  $$t_0 = x_0 = y_0 = \min\{x_1, y_1\} - 1$$

- and that

  $$V'(R \cup Q) = \{t_1, \ldots, t_{j-1}\}$$

  for some $j \geq 1$.

The merging process guarantees that

$$t_{j-1} \;=\; \max\{x_{l-1}, y_{h-1}\}$$

and chooses

$$t_j \;=\; \min\{x_l, y_h\} \tag{7.18}$$

Together with the implicit constraints that $x_{l-1} < x_l$ and that $y_{h-1} < y_h$, this implies

$$\begin{aligned} x_{l-1} \;&\leq t_{j-1} < t_j \leq\; x_l \\ y_{h-1} \;&\leq t_{j-1} < t_j \leq\; y_h \end{aligned}$$

Consequently, expression (7.17) can be applied to both $s'_R(x_l)$ and $s'_Q(y_h)$ when choosing t_j according to (7.18). Thus it is

$$\begin{aligned} s'_{R\cup Q}(t_j) \;=\; & \mathit{round}\left(\frac{t_j - t_{j-1}}{x_l - x_{l-1}} \cdot s'_R(x_l)\right) \;+ \\ & \mathit{round}\left(\frac{t_j - t_{j-1}}{y_h - y_{h-1}} \cdot s'_Q(y_h)\right) \end{aligned} \tag{7.19}$$

The modified version of the merge algorithm is shown in figure 7.18. Its general structure is the same as in the case of two complete IP-tables being merged. However, it uses (7.19) rather than equation (7.13).

7.5.3 Merging Complete and Incomplete IP-Tables

There might be a situation that several types of IP-tables are used within a temporal database. In this case, one can expect complete and incomplete IP-tables to be merged at some stage. This cannot be done by using one of the algorithms that have been discussed in sections 7.5.1 and 7.5.2 but by a hybrid one that uses parts of both. Figure 7.19 shows the algorithm for the case that a complete IP-table $I(R)$ is merged with an incomplete IP-table $I'(Q, b)$[6]. The result is necessarily an incomplete IP-table because the information of $I'(Q, b)$ is already incomplete. Therefore notations like $s'_{R\cup Q}$, $V'(R \cup Q)$ and N' are used rather than $s_{R\cup Q}$, $V(R \cup Q)$ or N. The algorithm is structured similarly to the ones in figures 7.17 and 7.18 and applies (7.17) only to $I'(Q, b)$.

[6] As mentioned earlier, we use the notation for condensed IP-tables in this case although the techniques apply to endpoint IP-tables too.

/* Merge two condensed (or endpoint) IP-tables $I'(R,a)$ and $I'(Q,b)$ */
/* $V'(R,a) = \{x_1, \ldots, x_A\}$ with $x_{l-1} < x_l$ for $l = 2, \ldots, A$ */
/* $V'(Q,b) = \{y_1, \ldots, y_B\}$ with $y_{h-1} < y_h$ for $h = 2, \ldots, B$ */

$V'(R \cup Q) = \emptyset$
$t_0 = x_0 = y_0 = \min\{x_1, y_1\}$ /* for convenience */
$l = 1$ /* index of next element of $V'(R,a)$ to be merged */
$h = 1$ /* index of next element of $V'(Q,b)$ to be merged */
$j = 1$ /* index of next element of $V'(R \cup Q)$ to be created */

while $l \leq A$ **and** $h \leq B$ **do** /* merge tables */
  $o_{R\cup Q}[j] = o_R[l] + o_Q[h]$
  if $x_l = y_h$ **then**
    $s'_{R\cup Q}[j] = \mathit{round}((x_l - t_{j-1})/(x_l - x_{l-1}) \cdot s'_R[l]) + \mathit{round}((x_l - t_{j-1})/(y_h - y_{h-1}) \cdot s'_Q[h])$
    $V'(R \cup Q = V'(R \cup Q) \cup \{x_l\}$
    $l = l + 1$
    $h = h + 1$
  else if $x_l < y_h$ **then**
    $s'_{R\cup Q}[j] = \mathit{round}((x_l - t_{j-1})/(x_l - x_{l-1}) \cdot s'_R[l]) + \mathit{round}((x_l - t_{j-1})/(y_h - y_{h-1}) \cdot s'_Q[h])$
    $V'(R \cup Q = V'(R \cup Q) \cup \{x_l\}$
    $l = l + 1$
  else /* $x_l > y_h$ */
    $s'_{R\cup Q}[j] = \mathit{round}((y_h - t_{j-1})/(x_l - x_{l-1}) \cdot s'_R[l]) + \mathit{round}((y_h - t_{j-1})/(y_h - y_{h-1}) \cdot s'_Q[h])$
    $V'(R \cup Q = V'(R \cup Q) \cup \{y_h\}$
    $h = h + 1$
  fi
  $j = j + 1$
od

if $h > B$ **then**
  while $l \leq A$ **do** /* add rest of $V'(R,a)$ */
    $s'_{R\cup Q}[j] = s'_R[l]$
    $o_{R\cup Q}[j] = o_R[l]$
    $V'(R \cup Q = V'(R \cup Q) \cup \{x_l\}$
    $l = l + 1$
    $j = j + 1$
  od
else
  while $h \leq B$ **do** /* add rest of $V'(Q,b)$ */
    $s'_{R\cup Q}[j] = s'_Q[h]$
    $o_{R\cup Q}[j] = o_Q[h]$
    $V'(R \cup Q = V'(R \cup Q) \cup \{y_h\}$
    $h = h + 1$
    $j = j + 1$
  od
fi

$N' = j - 1$

Figure 7.18: The merge algorithm for incomplete IP-tables.

/* Merge a complete IP-tables $I(R)$ and a condensed one $I'(Q,b)$ */
/* $V(R)$ = $\{x_1, \ldots, x_A\}$ with $x_{l-1} < x_l$ for $l = 2, \ldots, A$ */
/* $V'(Q,b)$ = $\{y_1, \ldots, y_B\}$ with $y_{h-1} < y_h$ for $h = 2, \ldots, B$ */

$V'(R \cup Q) = \emptyset$
$t_0 = y_0 = \min\{x_1, y_1\}$ /* for convenience */
$l = 1$ /* index of next element of $V(R)$ to be merged */
$h = 1$ /* index of next element of $V'(Q,b)$ to be merged */
$j = 1$ /* index of next element of $V'(R \cup Q)$ to be created */

while $l \leq A$ **and** $h \leq B$ **do** /* merge tables */
  $o_{R\cup Q}[j] = o_R[l] + o_Q[h]$
  if $x_l = y_h$ **then**
    $s'_{R\cup Q}[j] = s_R[l]) + \mathit{round}((x_l - t_{j-1})/(y_h - y_{h-1}) \cdot s'_Q[h])$
    $V'(R \cup Q = V'(R \cup Q) \cup \{x_l\}$
    $l = l + 1$
    $h = h + 1$
  else if $x_l < y_h$ **then**
    $s'_{R\cup Q}[j] = s_R[l]) + \mathit{round}((x_l - t_{j-1})/(y_h - y_{h-1}) \cdot s'_Q[h])$
    $V'(R \cup Q = V'(R \cup Q) \cup \{x_l\}$
    $l = l + 1$
  else /* $x_l > y_h$ */
    $s'_{R\cup Q}[j] = \mathit{round}((y_h - t_{j-1})/(y_h - y_{h-1}) \cdot s'_Q[h])$
    $V'(R \cup Q = V'(R \cup Q) \cup \{y_h\}$
    $h = h + 1$
  fi
  $j = j + 1$
od

if $h > B$ **then**
  while $l \leq A$ **do** /* add rest of $V'(R,a)$ */
    $s'_{R\cup Q}[j] = s_R[l]$
    $o_{R\cup Q}[j] = o_R[l]$
    $V'(R \cup Q = V'(R \cup Q) \cup \{x_l\}$
    $l = l + 1$
    $j = j + 1$
  od
else
  while $h \leq B$ **do** /* add rest of $V'(Q,b)$ */
    $s'_{R\cup Q}[j] = s'_Q[h]$
    $o_{R\cup Q}[j] = o_Q[h]$
    $V'(R \cup Q = V'(R \cup Q) \cup \{y_h\}$
    $h = h + 1$
    $j = j + 1$
  od
fi

$N' = j - 1$

Figure 7.19: The algorithm for merging a complete and an incomplete IP-table.

7.6 Histograms and IP-Tables

Query optimisers in a DBMS employ statistical profiles of the data. Such statistical profiles are complex objects that contain quantitative descriptors. One form of descriptor is a *histogram*. They form part of the family of non-parametric methods to estimate the frequency distribution of attribute values [Mannino et al., 1988].

In order to elaborate similarities between IP-tables and histograms we want to introduce types of histograms by an example. Let us consider the distribution of values of an attribute 'age' in figure 7.20: the first column contains the attribute values, the second the frequency of that value as it appears in the 'age' attribute of some relation, and the third column contains the cumulative of the frequencies. Instead of storing the entire and precise frequency distribution (as shown in columns 1 and 2 of figure 7.20) it is reasonable to store the frequencies for ranges of values. The result of this process is called a *histogram*. Essentially, there are three types of histograms that have been proposed in the literature:

- **Equal-width histograms**
  These histograms use equally sized value ranges, i.e. value buckets of equal widths. Figure 7.21 shows an equal-width histogram for the example of figure 7.20. It uses ranges that comprise five of the original attribute values each.

- **Equal-height histograms**
  These histograms use ranges such that each frequency is the same. In the example of figure 7.20 there are 100 values. If we assume that we want to create four buckets then each bucket should contain 25 values. We can imagine that (theoretically) this is done by sorting the relation over the age attribute. Then four buckets are created by putting the first 25 tuples into the first bucket, the next 25 into the second and so forth. Thus the first bucket would have ages 20 to 28, the second would only have ages of 28, the third ages between 29 and 34 and the final one ages from 34 to 39 (see figure 7.22). For advantages of equal-height over equal-width histograms please refer to [Piatetsky-Shapiro and Connell, 1984] or [Mannino et al., 1988].

- **Variable-width histograms**
  Several researchers suggested that widths are set so that the values within

each bucket are approximately uniformly distributed. This improves the accuracy of the selectivity estimations. Figure 7.23 shows a variable-width histogram for the example of figure 7.20.

Now we want to compare histograms and IP-table. To that end, we can consider $s_R(t)$ as a frequency distribution of the startpoints of timestamp intervals occuring in relation R. $o_R(t)$ can be regarded as an overlap frequency distribution. In that sense, a complete IP-table corresponds to two frequency distributions (that of $s_R(t)$ and that of $o_R(t)$), similar to the one shown in figure 7.20 for the atomic age attribute. The condensation process for IP-tables compares to the creation of value ranges – these correspond to the collapsed timepoints – of equal widths as the same number of timepoints are collapsed into one each time. Accordingly, we sum up the frequencies $s_R(t)$ for the individual ranges. However, we cannot treat the $o_R(t)$ values as frequencies in this case. As we have seen in the discussion of condensation (section 7.3.3) we keep the $o_R(t)$ value for the maximum t value of each range.

In summary this means that there are obvious similarities between IP-tables and histograms. In fact, one can consider to apply some of the methods for condensing or compressing frequency distributions into histograms to (complete) IP-tables too. However, many of the compressing methods for histograms were designed having the usage of histograms for selectivity estimation in mind. The type of variable-width histogram that we described above is an obvious example for that. However, the main purpose of IP-tables – at least in the context of this work – is partitioning rather than selectivity estimation. Therefore one needs to consider carefully whether the compressing methods that are beneficial in the case of selectivity estimation are equally favourable in the case of partitioning. A second important issue is that ranges (or buckets) for histograms are created in order to compress one single frequency distribution. In the case of IP-tables one has to consider two frequency distribution which in itself have to be treated differently as outlined in the previous paragraph.

The relationship between IP-tables and histograms and the possible application of histogram techniques is certainly an interesting topic for future research. Analysing this relationship here would lead away from our main goal which is to investigate the suitability of IP-tables for efficiently supporting the optimisation of partitioned temporal join processing.

Recent years have seen an extension of histograms beyond atomic data types. One example is this use of histograms in the context of processing multimedia data, such as in [Gong et al., 1996] or [Ng and Tam, 1997]. Other devel-

Age	Number	Cumulative
20	2	2
21	3	5
22	5	10
23	8	18
24	2	20
25	0	20
26	0	20
27	0	20
28	30	50
29	2	52
30	8	60
31	5	65
32	5	70
33	0	70
34	10	80
35	14	94
36	2	96
37	2	98
38	1	99
39	1	100

Figure 7.20: An example of an attribute value frequency distribution.

opments focus on specific purposes for which histograms are used. This has led to a variety of histogram types which goes beyond the three basic types that we have outlined above. [Poosala et al., 1996] provide a taxonomy for previously and recently proposed histograms. Furthermore there are papers that look at several aspects of histogram processing, such as error propagation [Ioannidis and Christodoulakis, 1993] or histogram maintenance [Gibbons et al., 1997]. As we already mentioned above, we can expect several results of this research being valuable in the context of IP-tables too.

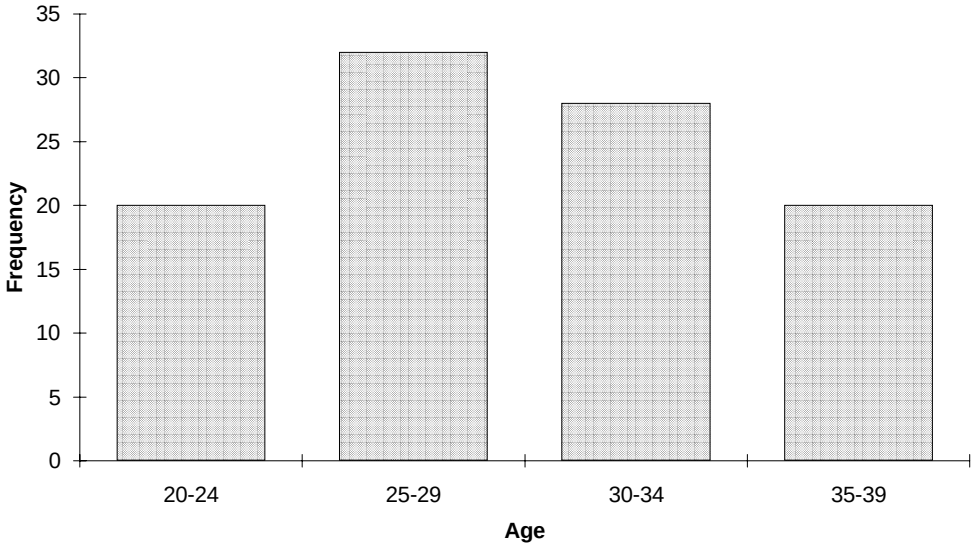


Figure 7.21: An equal-width histogram for the distribution of figure 7.20.

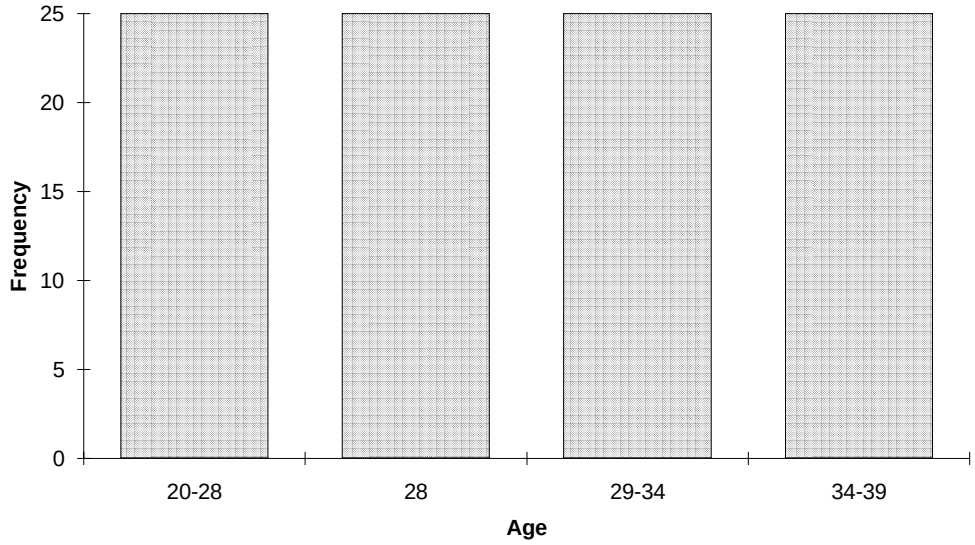


Figure 7.22: An equal-height histogram for the distribution of figure 7.20.

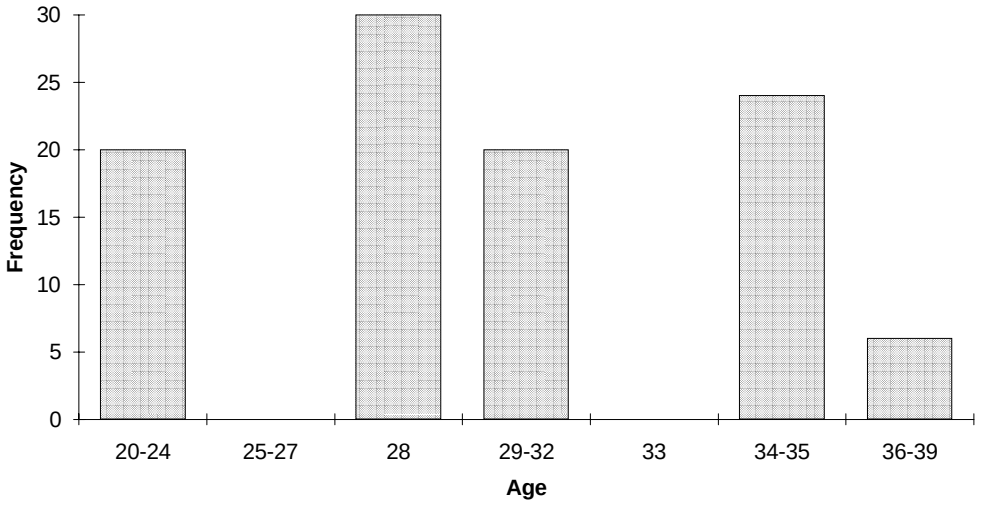
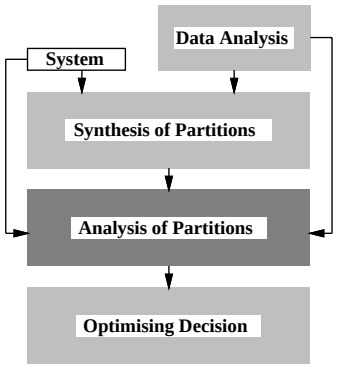


Figure 7.23: A variable-width histogram for the distribution of figure 7.20.

Chapter 8

Performance Model

After having discussed the initial stage of the optimisation process in chapter 7, we now want to move on to stage 3 in which the performance of a partition is determined. The reason for skipping stage 2 at this point is that we require a good and thorough understanding of the cost implications for designing partitioning strategies that are to be employed in stage 2. In order to create efficient partitions we need to know what the expensive parts of partitioned temporal join processing are and how these are influenced by the choice of a partition.



The purpose of this chapter is to create a performance model for partitioned temporal join processing. This model will not only enable us to design efficient partitioning strategies but it is also an integral part of the optimisation process: it is the optimiser's principle tool for deciding on the quality of a particular partition.

8.1 Outline

The task of creating an effective performance model is not straightforward because there are many factors that affect the costs of a temporal join. Amongst these are, for example, characteristics of the hardware architecture, issues of the (parallel) programming paradigm and the choice of the temporal join algorithm. If too many of these issues are incorporated into the model then it might be specific to one particular hardware architecture, one particular programming paradigm and/or one particular temporal join algorithm. On the other hand, if we omit or generalise too many of these issues then we prob-

ably miss out important factors that affect the join performance. This means that there is not *one, single* performance model which can be considered as appropriate but *many*. Our intention is to create one that seeks a good tradeoff between covering as many situations (with respect to hardware and software configurations) as possible while still being specific enough to achieve a reasonable performance prediction. In other words: it will necessarily be a compromise model. We divided the task of creating a performance model into three subtasks (see also figure 8.1):

- In section 8.2, we consider the *hardware* issues that affect the performance. An architectural model is presented. It is parameterised by two variables: depending on the values of these variables we get a single-processor architecture, a parallel shared-everything (SMP) architecture, a parallel shared-nothing architecture or a parallel hybrid, two-level architecture that incorporates elements of the shared-everything and the shared-nothing approaches. This covers a broad spectrum of possible architectures and therefore supports our goal to be as general as possible.

- In section 8.3, we look at the *software* aspect, i.e. the temporal join algorithm and how it works on the architectural model. In chapter 4, we have presented a wide range of temporal join algorithms of which only those are relevant that employ explicit and symmetric partitioning. Please recall that the performance of all the other temporal join algorithms is not affected by the choice of a partition. This still leaves us with a a variety of possible algorithms. We necessarily have to compromise here. However, this does not imply that we cannot create a model that provides a realistic feedback on the performance impacts of partitioning: we can pick an algorithm which represents a family of algorithms whose performances are similarly affected by the choice of a partition.

- The cost model for partitioned temporal join processing is derived in section 8.4. This model incorporates the assumptions and compromise decisions that have been taken in sections 8.2 and 8.3.

The chapter is concluded in section 8.5 where we evaluate the performance model on top of a uniform workload, i.e. we assume that the temporal intervals are uniformly distributed and of a uniform length. In this case a uniform partition of the data is optimal. Therefore we can draw conclusions about performance characteristics that are partition-independent. This will allow us

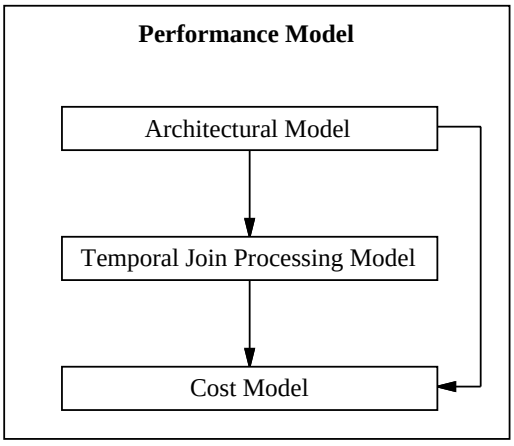
to draw a variety of partition-independent conclusions, in particular it will prove that the assumptions that we had to make about the underlying program paradigms only have minor impacts on the cost model.

This analytical way of modeling the performance has two major advantages in comparison to alternative approaches for determining processing costs, such as simulation or implementation on a real hardware platform:

1. It can be used not only for the evaluation of our techniques but provides a tool for an optimiser to estimate the performance on which it can base its decisions. Simulation or real implementations can only cater for the first purpose.

2. As already elaborated above, we want to obtain a model for a variety of hardware platforms. Implementations and simulations produce results that are specific to the respective hardware or range of hardware platforms.

These advantages are accompanied by the disadvantage that the absolute cost figures that are obtained probably do not compare directly with the ones that are achieved in reality. A simulation or an implementation of the operation would achieve preciser results. However, as we are concerned with comparing possible partitions in a platform-independent manner, we will use an analytical approach. In future research it might be interesting to validate our analytical model by simulating or implementing the operation.



An arrow indicates that one model influences the other.

Figure 8.1: The structure of the performance model and the modeling process.

8.2 The Architectural Model

8.2.1 Introduction

Nowadays, high-performance database management systems (DBMS) are running on a variety of hardware platforms. There are two categories:

- single processor servers
- multiprocessor servers

Machines of the first category usually employ a single, but very powerful processor. Although there are still many DBMS installations running on uniprocessors, the use of multiprocessor systems is vital for performance whenever the database size or the workload cause the CPU of a uniprocessor system to be the performance bottleneck. Multiprocessor servers combine the raw computing power of many (commodity) processors in order to achieve high performance. However, parallelism is not restricted to the CPU but also to I/O and main memory access. There are many ways in which processors, disks, memory modules, buses etc. can be combined in order to build a parallel database server and this section will discuss some of the resulting architectural categories.

At the end of the 1980s and the beginning of the 1990s there was a wide and controversial discussion[1] about the question "Which is the most suitable parallel architecture to support parallel database systems?" It was expected that one could draw conclusions on the system's performance by analysing its underlying architecture. For a while, there was a confusion about what the term "architecture" actually comprised: only the system's hardware or also the software? Therefore many researchers mixed hard- and software aspects within this discussion. Actually, this was not a problem as the first parallel database system prototypes used to have matching hard- and software architectures. However, things changed when parallel DBMS technology started to be commercially exploited.

In the last few years, many vendors have tried to make their parallel DBMS products independent from specific parallel hardware platforms in order to achieve a wider acceptance in the market of high-end DBMS products. This resulted in the fact that a DBMS's software architecture does not necessarily match the underlying hardware architecture. Similarly, vendors of parallel

[1] See summary in section 8.2.2.

hardware moved to general-purpose architectures that can run software of any type but with certain software architectures being more favourable than others.

This development made it even more difficult to predict a system's performance from an analysis of the underlying architectures. Alternatives were proposed such as the 5-layer-model by Norman and Thanisch. They suggest to base a performance analysis on 5 layers with each layer representing a system's hard- and software components (see figure 8.2). Lines between the components describe dataflows. By describing a system on top of this model one can now see in which way workloads are balanced between the components within the system [Norman and Thanisch, 1995].

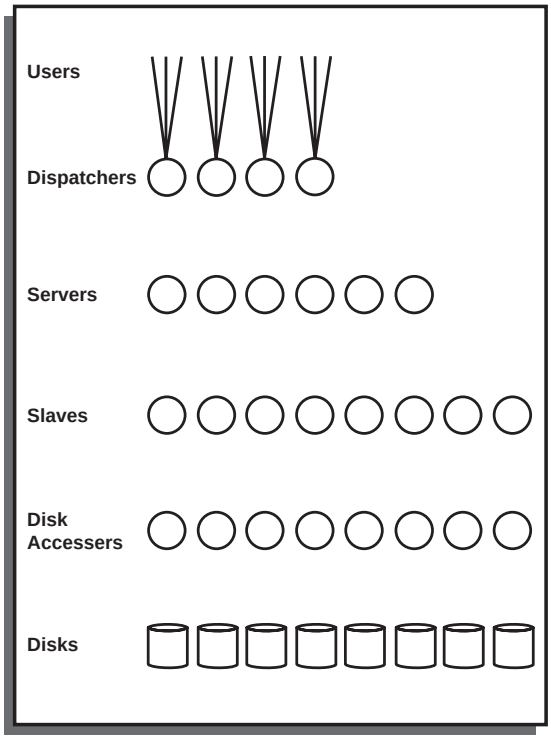


Figure 8.2: The 5 layers of the generic model. Source: [Norman and Thanisch, 1995].

A further issue, which makes performance modeling a difficult task, is the following: practical experience shows that a system's performance is quite often also a result of tuning, i.e. the proper configuration of the hardware with respect to the software and the workload and the configuration of the software with respect to the hardware and the workload. Tuning is an important issue as practical evidence shows that there is a huge difference in performance between a well-tuned and a poorly-tuned system [Witkowski, 1993].

As a conclusion from the above, it becomes obvious that it is a difficult and complex task to determine a system's performance; there is a huge number

of factors and facts to be considered. However, in this thesis we are not concerned with the overall performance of a DBMS under a certain workload but with the performance of one particular operation. Consequently, we do not need to make assumptions about the system's software architecture; we can merely concentrate on its hardware. This might stand in contrast to what we said above but is purely justified by the fact that we concentrate on one single operation rather than an entire DBMS.

This allows us to make some simplifying assumptions: we perceive the system environment as a set of hardware resources that are available for processing the temporal join. These resources are characterised by parameters, e.g. the (current) amount of free memory, the (current) communication bandwidth, the number of processing nodes that are available for processing the join at that particular moment etc. We assume these parameters to be dynamic, i.e. they describe the *current* potential of the system, rather than static, i.e. they are not supposed to be constant all the time. In that way, we incorporate the system's load / workload without making any assumptions about it. A high workload, for example, might imply a small amount of free memory, low communication and I/O bandwidths etc., whereas low workloads imply more beneficial parameter values. What remains to be defined is how these components interact, i.e. the hardware architecture.

In section 8.2.2, we summarise the architectural discussion that was mentioned earlier. It provides an overview over the various basic architectural types that can be considered. The arguments in favour and against these basic types explain the convergence to hybrid architectures which are presented in section 8.2.3. The latter incorporate concepts of various basic types. We pick one of these architectures to be the one on which our performance model is based. As outlined in section 8.1, it is parameterised by two variables. These parameters provide us with the flexibility to setting up either a single-processor architecture, a parallel shared-memory (SMP) architecture, a parallel shared-nothing architecture or a hybrid, two-level architecture incorporating advantages of the shared-memory and the shared-nothing approaches.

8.2.2 Summary of the Architectural Discussion

Traditionally, architectures for parallel DBMS were categorised by the way in which processors share hardware resources like disk devices and memory. This categorisation initially appeared in [Stonebraker, 1986] and was mainly meant to be a discussion around the most appropriate parallel hardware archi-

tecture. Many researchers have participated in this discussion in the following years; see e.g. [Bhide and Stonebraker, 1988], [DeWitt and Gray, 1990], [Hua et al., 1991], [DeWitt and Gray, 1992], [Bergsten et al., 1993], [Valduriez, 1993b], [Baru et al., 1995], [Gray, 1995] and many others, base their arguments on it. In this section, we briefly describe the architectural categories and summarise the conclusions that have been drawn. We note that many arguments that were brought forward are of a historical nature because they reflect on the state of the technology in the late 1980s and early 1990s and do not consider recent developments. The categories are:

- shared-memory,
- shared-disk,
- shared-nothing.

Shared-Memory

Shared-memory (SM) – some authors, like [Hua et al., 1991] or [Bergsten et al., 1993], prefer the equivalent term *shared-everything* – means that all disks and all memory modules are shared by the processors as shown in figure 8.3. This means that all disks are equally accessible by all processors and that there is a global address space for main memory. The latter can be implemented as a physically distributed memory in which each processor has a local memory which forms a part of the global memory[2]. There are two forms for accessing this distributed shared memory [Tannenbaum, 1994]:

- There is a *uniform access memory* (UMA) in which uniformity is guaranteed by a hardware-driven caching mechanism. This means that, in theory, accesses to any location in memory are at the same costs. In practice, however, caching cannot entirely extinguish the difference between local and remote memory access costs but it makes this difference bearable. The caching hardware, however, is complex and expensive and limits SM-architectures to a small number of processors.
- The alternative is a *non-uniform memory access* (NUMA) in which access to local memory is typically 10 times faster than to a remote access to an address space which is located at another node. Remote accesses are avoided either by the software or by the operating system which aims

[2]See discussion about shared-disk.

to shift memory pages to the location from which it is most frequently accessed. In contrast to UMA, the NUMA approach is scalable, much cheaper and more flexible at the expense of leaving the memory access optimisation to the operating system.

A symmetric multiprocessor (SMP) is an example for the shared-memory concept: it integrates a small number of identical processors in order to combine their raw computing power. These processors cooperate over a single memory.

The following arguments have been raised when discussing SM-architectures: It is said that SM is simple to program, essentially because of the global address space in main memory. Load balancing can be arranged relatively easily because each processor has equal access to all disks. Communication among the processors is fast (and incurs low overhead) as they can cooperate via main memory. However, system costs are high for big systems because the bus becomes a bottleneck and various hardware mechanism have to be employed to tackle this problem. Conflicting accesses to main memory can decrease the performance. It is also argued that access to main memory is the reason why SM-architectures do not scale up very well: [Bhide and Stonebraker, 1988] showed that beyond a certain number of processors, access to main memory can become a bottleneck that limits the system's processing speed. SM systems are therefore limited to a small number of processors ([Valduriez, 1993a] mentions 20; [Baru et al., 1995] argues that the limit is around 10 RISC System / 6000 processors).

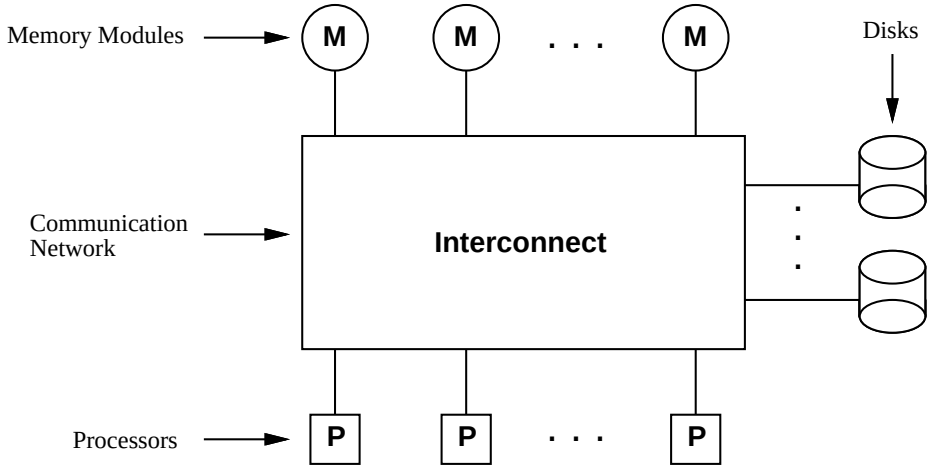


Figure 8.3: Shared-Memory Architecture

Shared-Disk

In a *shared-disk* (SD) system, each processor has its private memory. The access to disks, however, is shared by all of them. Figure 8.4 shows this architecture. Actually it shows the way in which the SM is frequently implemented, namely as a *distributed* shared memory with each processor holding one part of the global memory. For that reason SD and SM can be considered as synonymous nowadays.

In the following, we summarise the arguments that have been brought up in favour or against SD systems: It is argued that the costs for SD system are relatively low as the interconnect could be a bus system based on standard technology. It is also argued that load balancing is also relatively easy for the same reason as in the SM case, and that the availability of data is higher than in an SN system (see below) as a crash on one processor does not result in the data of a particular disk being unavailable. Software from uniprocessor systems can be easily migrated since the data on disk need not be reorganised [Valduriez, 1993a]. Much of the down-side of SD systems is said to relate to an increase in complexity, e.g. caused by the cache coherency control mechanisms that are necessary to maintain consistent disk pages in the processors' individual caches. A centralised lock management is also required. All this limits the scalability of a SD system. Finally, the access to the shared disks might result in a bottleneck through a limited interconnect capacity.

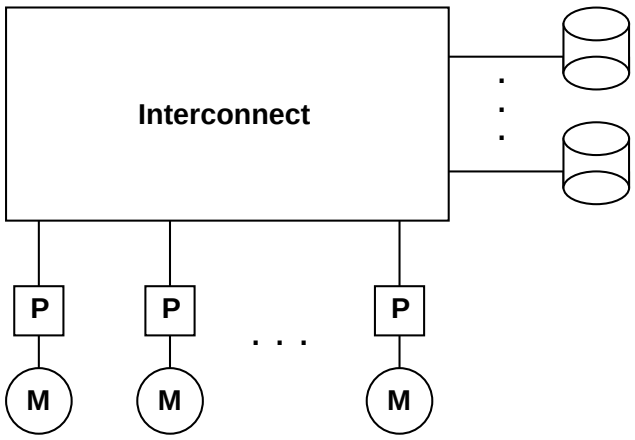


Figure 8.4: Shared-Disk Architecture

Shared-Nothing

In a *shared-nothing* (SN) system, each processor has its private memory and has at least one disk connected; the processor acts as a server for the data on this disk. Figure 8.5 shows this type of architecture.

The arguments around the SN-architecture are as follows: It is said that the costs of a SN system are low because it can essentially be constructed from commodity components[3]. Theoretically, SN can scale and speed up linearly; in practice it is argued that the interconnect becomes saturated beyond a certain volume of communication. Availability is also often considered to be a serious problem. Load balancing is another problem: it is argued that data skew can cause serious imbalances. Furthermore, load imbalance can be caused by the the execution of operations being in some way predetermined by the data placement on the disks and the necessity to avoid huge data shipping through the network to other processors.

For a long time, SN has been considered as the best parallel architecture, mainly because of its allegedly unlimited scalability. There are two reasons why this advantage has not manifested itself in practice:

- As already mentioned above, the interconnect gets saturated beyond a certain point. Academics often pointed to Teradata machines[4] as an example of a commercial SN product that would allegedly scale up to systems with "over 1000 processors" [DeWitt and Gray, 1992]. However, Teradata itself admitted that its interconnect, the YNET, would not scale to the maximum, physically feasible configuration of 1000 processors [Witkowski, 1993].

- As a matter of fact, there is a trend which offers an alternative to scalability: processor speed doubles roughly every 18 months [Gray, 1995]. Therefore, instead of adding *quantity* (i.e. more processors) one can add *quality* (i.e. faster processors), thus avoiding the problem of the interconnect saturation. We used the workload and performance model provided in [Hua et al., 1991] to compare these two possibilities with respect to join processing times. Starting with an initial SN architecture with 10

[3]This argument his entirely historical as many SM systems are now entirely built of commodity parts whereas SN systems frequently have proprietary (and therefore expensive) interconnect.

[4]The interested reader might refer to papers such as [Teradata Corporation, 1983], [Teradata Corporation, 1985], [Carino and Kostamaa, 1992], [Sloan, 1992], [Witkowski, 1993] and many others to get details about Teradata machines and their successors.

processors we multiplied the number of processors by a scaling factor $x = 2, \ldots, 5$. Using the same initial architecture we did the same experiment but this time we magnified the overall computing power by using processors that were $x = 2, \ldots, 5$ times faster. The result can be seen in figure 8.6 and proves that adding faster processors is preferable. This means that in practice scalability is actually a characteristic that is not as important as it is frequently claimed by many academic researchers. This also explains the fact that there are so few parallel computer systems on the market nowadays that employ a large number of simple processors – such as the Connection Machine [Hillis, 1985] or the MasPar MP-1 [Blank, 1990] – but a relatively small number – e.g. the Cray T3D [Cray Research, 1993] – or a handful – e.g. servers such as Sun's SPARCcenter 2000 and derived products [Cekleov et al., 1993] – of powerful processors.

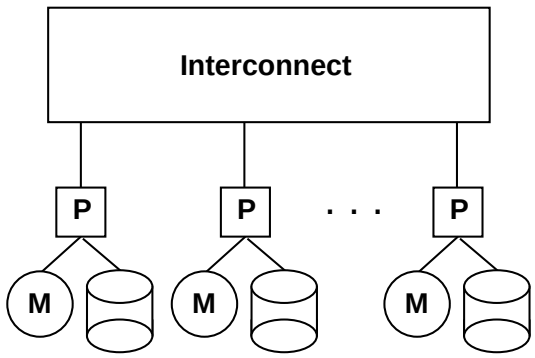


Figure 8.5: Shared-Nothing Architecture

8.2.3 A Hybrid Architecture

Regarding the characteristics of the preceding three architectural types it becomes obvious that there is no ideal, single consensus architecture for parallel database systems. Hua *et al.* proposed an architecture that combines the advantages of shared-nothing (scalability) with those of shared-memory (fast communication, easy load-balancing) [Hua et al., 1991]. $\mathcal{M}$ symmetric multiprocessor (SMP) nodes, each of which comprising $\mathcal{N}$ processors, are connected in a shared-nothing manner. The architecture is shown in figure 8.7. Many recent commercial products adopted this or similar architectures, such as the one outlined in figure 8.8: basically it is the same architecture as in figure 8.7 but

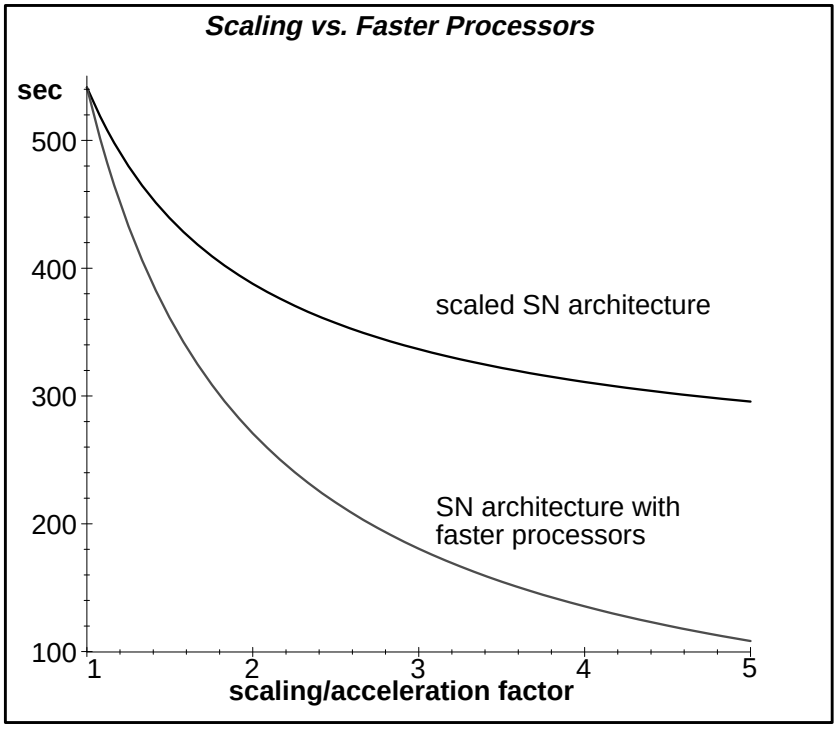


Figure 8.6: Scaling vs. using faster processors in a SN architecture.

allows access to disks from more than one node. As well as flexibility, this provides a certain redundancy in case that a node fails.

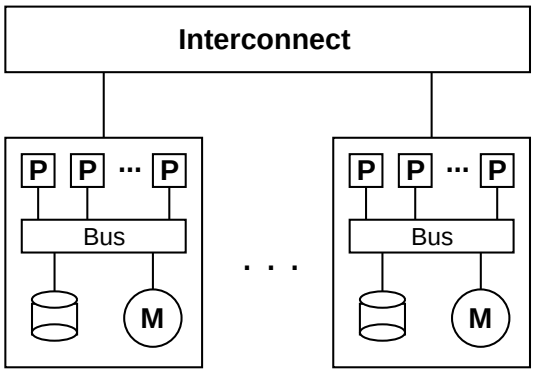


Figure 8.7: Hybrid architecture described in [Hua et al., 1991].

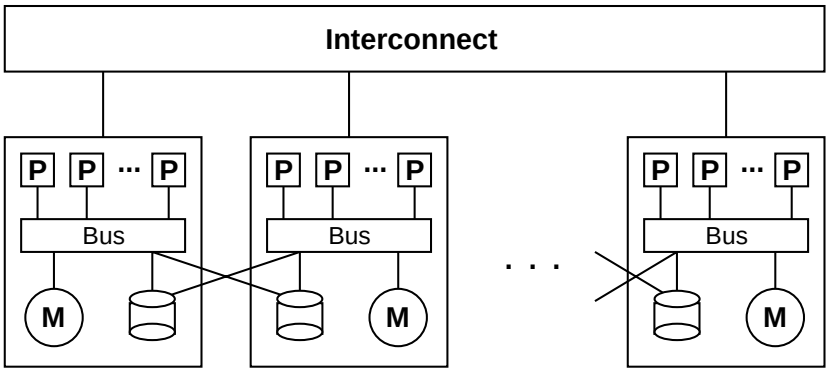


Figure 8.8: Hybrid architecture adopted by many recent commercial products

We will use this basic architectural model for modeling the performance of temporal joins. Hua *et al.* have proved its suitability for this purpose when analysing parallel join performances on this architecture and accurately simulating and predicting many architectural effects. We have already seen one example in figure 8.6. But this architectural model provides further advantages: the parameters $\mathcal{M}$ and $\mathcal{N}$ can be used to set up any of the architectural types that have been discussed so far:

- $\mathcal{M} = 1, \mathcal{N} = 1$: In this case, we have one processing node that contains one processor, i.e. a single-processor machine as it can be found in mainframes or other database servers.

- $\mathcal{M} = 1, \mathcal{N} > 1$: This is a SM-/SD-architecture: one SMP node comprising $\mathcal{N}$ processors.

- $\mathcal{M} > 1, \mathcal{N} = 1$: This is a SN-architecture with $\mathcal{M}$ single-processor nodes. Alternatively, it can be considered as a NUMA-based SM-/SD-architecture in which the interconnect represents a bus over which processors access non-local memory modules.

In the following sections, we will not assume any particular values for $\mathcal{M}$ and $\mathcal{N}$. We will use them as parameters that describe the architecture. Only in the experiments in section 8.5 and chapter 10 appropriate values will be chosen.

8.3 Temporal Join Processing Model

8.3.1 Preliminaries

Before we can start to describe how a temporal join $R \bowtie_C Q$ is processed, we have to lay out the starting situation that we assume when such a join is processed on a hardware architecture as in figure 8.7.

Firstly, we have to decide on the temporal join algorithm that is to be used. We will model the performance of the algorithm presented in section 4.4.3 for the following reasons:

- It is suitable for any configuration of the hybrid architecture that has been described in section 8.2.3; the algorithm presented by Soo *et al.*, for example, is restricted to sequential processing (see section 4.4.4).

- It is based on a symmetric partitioning approach and thus easily applicable to n-way joins for $n \geq 3$, too. In sections 4.4.5 and 4.5 we have already seen that Lu *et al.*'s spatially partitioned join can be expected to perform poorly in such a situation.

- It performs better than the basic algorithm of section 4.4.2 as it avoids redundancies.

Therefore it is the most versatile of the algorithms that are based on explicit partitioning. Furthermore, it can be expected to perform well according to the analysis of section 4.5.

Secondly, the temporal relations R and Q are assumed to be physically distributed over the disk systems of the $\mathcal{M}$ nodes: the disks of node i hold frag-

ments $\hat{R}_i$ and $\hat{Q}_i$ of R and Q respectively ($i = 1, \ldots, \mathcal{M}$) such that

$$\begin{aligned} R &= \bigcup_{i=1}^{\mathcal{M}} \hat{R}_i \\ Q &= \bigcup_{i=1}^{\mathcal{M}} \hat{Q}_i \end{aligned}$$

These fragments are supposed to be pairwise disjoint, i.e.

$$\begin{aligned} \hat{R}_i \cap \hat{R}_j &= \emptyset \\ \hat{Q}_i \cap \hat{Q}_j &= \emptyset \end{aligned}$$

for $1 \leq i < j \leq \mathcal{M}$. Later, we see that the first processing stage converts these initial fragments into the R'_k, R''_k, Q'_k, Q''_k for $k = 1, \ldots, m$, which are required for processing $R \bowtie_C Q$ based on symmetric partitioning. We will refer to this as the *repartitioning stage* (see figure 8.9).

Thirdly, there is a partitioning strategy which produces a partition $P = \{p_1, \ldots, p_{m-1}\}$ which is defined as in chapter 5. It is used within the repartitioning stage for creating the fragments R'_k, R''_k, Q'_k, Q''_k for $k = 1, \ldots, m$. Please note that it is P that provides the parameter m.

The repartitioning stage is followed by a *joining stage* in which the partial joins are computed. We assume that

$$R'_k \bowtie_C Q''_k, \quad R'_k \bowtie_C Q'_k, \quad R''_k \bowtie_C Q'_k \tag{8.1}$$

are processed sequentially on one processor as proposed in section 4.4.3. We refer to the computation of the three joins in (8.1) as RQ_k ($k = 1, \ldots, m$).

The machine has $\mathcal{M}$ nodes each with $\mathcal{N}$ processors. The nodes are numbered from 1 to $\mathcal{M}$ and the processors from 1 to $\mathcal{MN}$ such that all processors of node i have numbers from $(i-1)\mathcal{N}+1$ to $i\mathcal{N}$. A function *node*(j) gives the number of the node to which processor j belongs

$$\textit{node}(j) \;=\; (j \;\textit{div}\; \mathcal{M}) + 1$$

So there are $\mathcal{MN}$ processors for performing m computations RQ_k with $k = 1, \ldots, m$. We now look at the way in which the RQ_k will be distributed over the processors.

If $m \leq \mathcal{MN}$ then processors $1, 2, \ldots, m$ perform one computation RQ_k (for $k \in \{1, \ldots, m\}$) each. Processors $m+1, m+2, \ldots, \mathcal{MN}$ remain idle in that case. If $m > \mathcal{MN}$ then the workload distribution is as follows: the $\mathcal{MN}$ processors

are divided into two sets. One has A processors, each of which performs α computations RQ_k (for $k \in \{1, \ldots, m\}$) with

$$\alpha \;=\; \left\lceil \frac{m}{\mathcal{MN}} \right\rceil \tag{8.2}$$

The other $B = \mathcal{MN} - A$ processors perform β computations RQ_k (for $k \in \{1, \ldots, m\}$) with

$$\beta \;=\; \left\lfloor \frac{m}{\mathcal{MN}} \right\rfloor \tag{8.3}$$

This is illustrated in figure 8.10. Please note that the processors work concurrently but each processor processes its 'load' sequentially.

The values of A and B can easily be computed from the values of α and β from the following two constraints:

$$A + B \;=\; \mathcal{MN} \tag{8.4}$$

$$A \cdot \alpha \;+\; B \cdot \beta \;=\; m \tag{8.5}$$

From (8.4) follows that $B = \mathcal{MN} - A$. If this is used to replace B in (8.5) then we get

$$A \;=\; \frac{m - \beta \cdot \mathcal{MN}}{\alpha - \beta} \tag{8.6}$$

But if m is not a multiple of $\mathcal{MN}$ then we have

$$\alpha - \beta \;=\; 1$$

Therefore (8.6) works out to be

$$A \;=\; m - \beta \cdot \mathcal{MN} \tag{8.7}$$

and consequently

$$B \;=\; (\beta + 1) \cdot \mathcal{MN} - m \tag{8.8}$$

because of (8.4). If m is a multiple of $\mathcal{MN}$ then we have $\alpha = \beta$ and we do not need to divide the processors, i.e. it is $A = \mathcal{MN}$ and $B = 0$ in this case. Actually, we do not need to separate a situation with $m < \mathcal{MN}$ from one with $m > \mathcal{MN}$ as it is $\alpha = 1$ and $\beta = 0$ in that case which leads to $A = m$ and $B = 0$ because of (8.7) and (8.8) respectively. And if $m = \mathcal{MN}$ then it is $\alpha = \beta = 1$ which leads to $A = m$ and $B = 0$, too.

Now, we look at the opposite side of the coin, i.e. we want to determine the number j of the processor that performs computation RQ_k with $k \in \{1, \ldots, m\}$.

To that end we have to assume that the RQ_k are assigned to the processors as in figure 8.10. We note that this might not be the optimal assignment, for example if the most expensive computations coincide to hold subsequent numbers and therefore happen to be performed subsequently by the same processor. This causes the respective processor's load to be the most expensive one and the one that determines the overall join processing performance. In order to keep our performance model simple and manageable for a query optimiser, we do not optimise such a situation at this stage by rearranging the computations' order. Such a rearrangement could imply overhead costs if an optimal placement cannot be determined beforehand. This means that data might have to be transferred through the network and one would need to analyse the tradeoff between this overhead and the optimised costs in order to see if such a rearrangement was worth while.

The function *processor*(k) is used to give the number $j \in \{1, \ldots, \mathcal{MN}\}$ of the processor that performs computation RQ_k with $k \in \{1, \ldots, m\}$ according to an assignment as in figure 8.10. If RQ_k is among the first $A\alpha$ computations, i.e. $1 \leq k \leq A\alpha$, then it is performed by processor

$$\left\lceil \frac{k}{\alpha} \right\rceil$$

Similarly, if $k > A\alpha$ then it is performed by a processor $j > A$, i.e. by processor

$$\left\lceil \frac{k - A\alpha}{\beta} \right\rceil + A$$

This leads to *processor*(k) being defined as

$$\textit{processor}(k) \;=\; \begin{cases} \left\lceil \frac{k}{\alpha} \right\rceil & \text{for } 1 \leq k \leq A\alpha \\ \left\lceil \frac{k - A\alpha}{\beta} \right\rceil + A & \text{for } A\alpha < k \leq m \end{cases} \tag{8.9}$$

$L(R \cup Q) = [t_{\min}, t_{\max}]$ is the lifespan covered by the tuples of R and Q. In this context $t_{\min}$ and $t_{\max}$ are

$$\begin{aligned} t_{\min} &= \min\{L(R), L(Q)\} \\ t_{\max} &= \max\{L(R), L(Q)\} \end{aligned}$$

As mentioned above, P is an m-way partition of $L(R \cup Q)$, i.e. a set of $m - 1$ breakpoints

$$\{p_1, \ldots, p_{m-1}\}$$

with $p_k \in L(R \cup Q)$ for $k = 1, \ldots, m-1$. $p_0 = t_{\min} - 1$ (or $-\infty$) and $p_m = t_{\max}$ (or $+\infty$) are used as the left and the right delimiters of the plot. P divides $L(R \cup Q)$ into m partition ranges

$$(p_{k-1}, p_k] \;=\; \{t \in L(R \cup Q) : p_{k-1} < t \leq p_k\}$$

for $k = 1, \ldots, m$. The function $\mathit{fragment}_P(t)$ determines the number of the fragment (partition range respectively) to which a timepoint $t \in L(R \cup Q)$ belongs with respect to partition P

$$\mathit{fragment}_P(t) \;=\; k \text{ iff } t \in (p_{k-1}, p_k]$$

This means that a tuple r with timestamp $[t_s, t_e]$ is put into $R'_{\mathit{fragment}_P(t_s)}$ and the R''_k with the k given by the set

$$K''_P(r) \;=\; \{k : \mathit{fragment}_P(t_s) < k \leq \mathit{fragment}_P(t_e)\}$$

We will use the functions $\mathit{first}(j)$ and $\mathit{last}(j)$ to refer to the index of the first and the last computation that is performed on processor j. According to figure 8.10, these functions are defined as

$$\begin{aligned}
\mathit{first}(j) &= \begin{cases} (j-1)\alpha + 1 & \text{if } j \leq A \\ A\alpha + (j - A - 1)\beta + 1 & \text{if } j > A \end{cases} \\
\mathit{last}(j) &= \begin{cases} j\alpha & \text{if } j \leq A \\ A\alpha + (j - A)\beta & \text{if } j > A \end{cases}
\end{aligned}$$

Thus the first and last computations at a node i are

$$\begin{aligned}
\mathit{first\text{-}node}(i) &= \mathit{first}((i-1)\mathcal{N} + 1) \\
\mathit{last\text{-}node}(i) &= \mathit{last}(i\mathcal{N})
\end{aligned}$$

respectively. Later, it will be useful to refer to the first fragments R'_k, R''_k, Q'_k, Q''_k on each node, i.e. the first fragments on the first processor of this node. The indices k of these fragments are collected in the set K_{first}:

$$K_{\mathit{first}} \;=\; \{k : k = \mathit{first\text{-}node}(i) \;\wedge\; 1 \leq i \leq \mathcal{M}\}$$

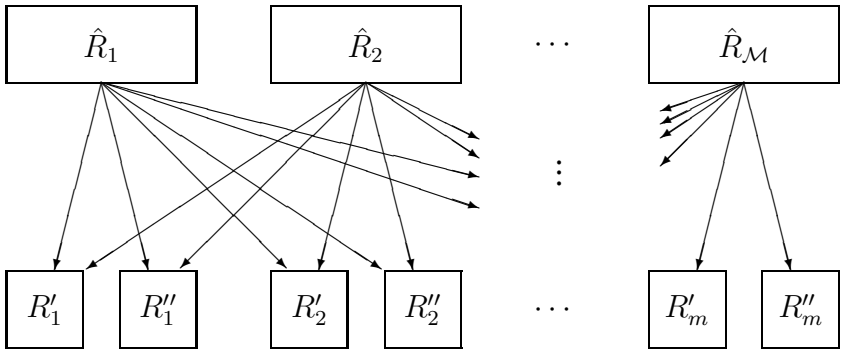


Figure 8.9: Repartitioning of the $\hat{R}_1, \ldots, \hat{R}_{\mathcal{M}}$.

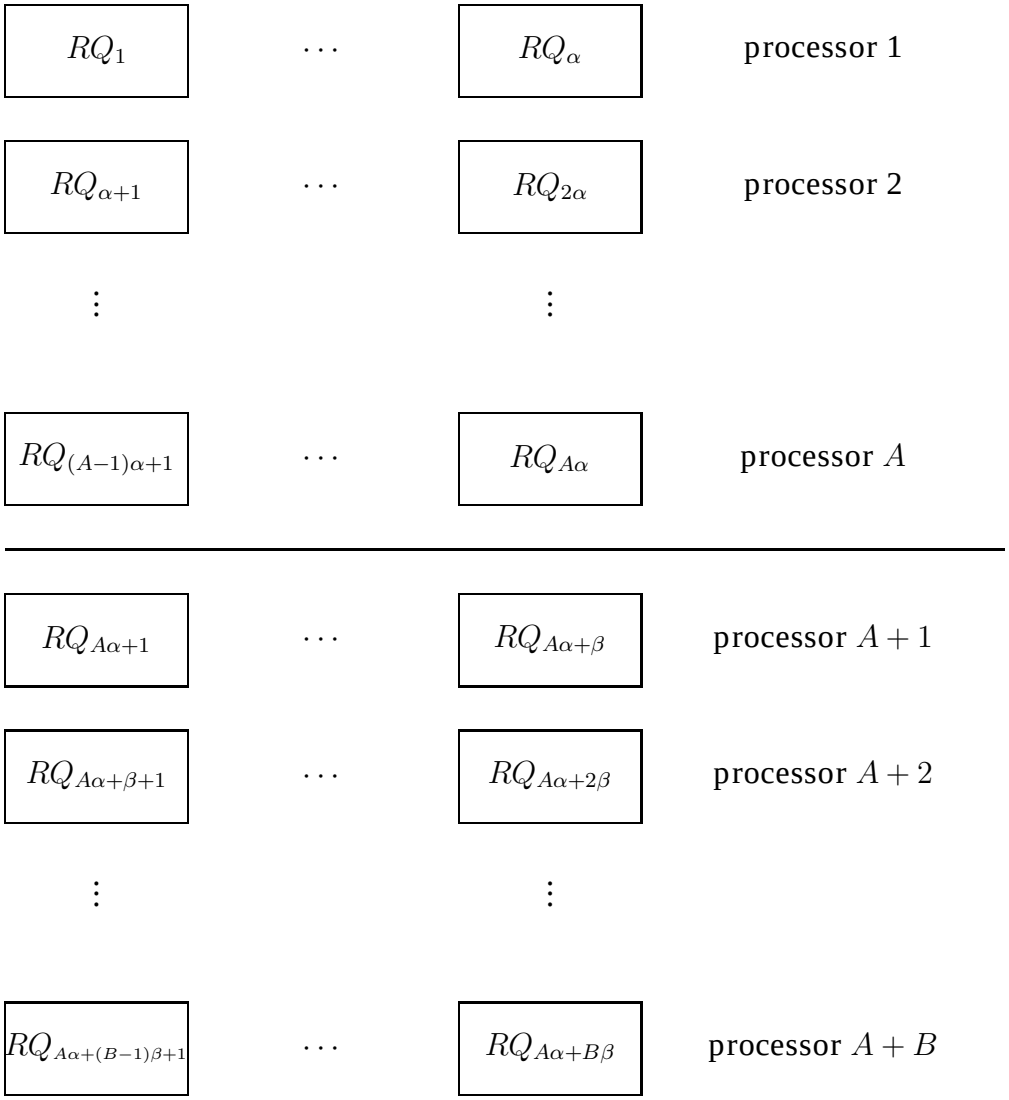


Figure 8.10: Workload distribution among processors.

8.3.2 Temporal Join Processing

In the previous section, we gave a rough outline of how a temporal join is going to be processed on the architectural model. Here, we give a precise, stepwise description of this process. It is based on the symmetric partitioning that was discussed in section 3.5.2. Thus the process comprises the following three stages (see figure 3.19):

1. A *(re-)partitioning stage* in which fragment $\hat{R}_1, \ldots, \hat{R}_{\mathcal{M}}$ are repartitioned to create fragments $R'_1, R''_1, \ldots, R'_m, R''_m$. The same is done for the $\hat{Q}_1, \ldots, \hat{Q}_{\mathcal{M}}$. See figure 8.9.

2. A *joining stage* in which the RQ_k are computed for $k = 1, \ldots, m$. See figure 8.10.

3. A *merging stage* in which the partial results are merged and written to disk.

In the remainder we will concentrate on the stages 1 and 2. Stage 3 might be omitted if further processing requires the data to be partitioned. Furthermore, from the performances comparison's point of view, one can argue that the time spent on stage 3 is (more or less) the same for all cases as it only consists of writing the join result (which is the same in all cases) to disk. It would only increase an already existing difference if some computations have already been off balance and have caused big partial results. But such an 'off-balance-situation' would already be penalised by a poor performance in stage 2, the stage that creates the 'off-balance-sized-results'. Therefore, stage 3 would only contribute marginal differences in the overall performance. Consequently, we can concentrate on the costs of stages 1 and 2 in order to create a performance model that allows us to *compare* computations using different partitions. In the remainder, we look at the processing within these two stages.

8.3.3 Stage 1: Repartitioning

In stage 1, the initial fragments $\hat{R}_i$ of R and $\hat{Q}_i$ of Q ($i = 1, \ldots, \mathcal{M}$) are repartitioned as illustrated in figure 8.9. On the architectural model, this process is performed as follows – the description is restricted to repartitioning of R; the process works analogously for Q.

(a) Each node i reads its fragment $\hat{R}_i$ of R; then each processor of that node processes the $\mathcal{N}$-th part of this fragment ($i = 1, \ldots, \mathcal{M}$).

(b) Each processor j has $2m$ hash buffers: $\mathcal{B}'_{j,1}, \ldots, \mathcal{B}'_{j,m}$ to accommodate tuples[5] for $R'_1, \ldots, R'_m$ and hash buffers $\mathcal{B}''_{j,1}, \ldots, \mathcal{B}''_{j,m}$ for tuples for $R''_1, \ldots, R''_m$.

Furthermore there are $2m$ output buffers $\mathcal{O}'_1, \ldots, \mathcal{O}'_m, \mathcal{O}''_1, \ldots, \mathcal{O}''_m$. They are distributed over all $\mathcal{MN}$ processors of the machine according to figure 8.10, i.e. $\mathcal{O}'_k$ and $\mathcal{O}''_k$ are located at processor $j = \mathit{processor}(k)$, or, put the other way round, a processor j has the output buffers $\mathcal{O}'_{\mathit{first}(j)}, \ldots, \mathcal{O}'_{\mathit{last}(j)}$ and $\mathcal{O}''_{\mathit{first}(j)}, \ldots, \mathcal{O}''_{\mathit{last}(j)}$ (see figure 8.11). An output buffer $\mathcal{O}'_k$ will later receive all the tuples of R'_k which come from the hash buffers $\mathcal{B}'_{j,k}$ for all $j = 1, \ldots, \mathcal{MN}$. Similarly, $\mathcal{O}''_k$ receives all the tuples of R''_k which come from hash buffers $\mathcal{B}''_{j,k}$ for all $j = 1, \ldots, \mathcal{MN}$.

A processor j hashes its tuples to the hash buffers in the following way: a tuple r with timestamp $[t_s, t_e]$ is put into

(i) hash buffer $\mathcal{B}'_{j,k}$ with $k = \mathit{fragment}_P(t_s)$

(ii) hash buffers $\mathcal{B}''_{j,k}$ with $k \in K''_P(r) \cap K_{\mathit{first}}$.

Step (i) puts the tuple in the fragment that covers the range in which the timestamp's startpoint t_s falls; step (ii) puts the tuples in those fragments R''_k that are processed by the first processor on nodes (other than that covered by step (i)) that will perform an RQ_k that involves r. By doing so, we avoid the situation in which more than one copy of r is sent to the same node over the interconnect. Within a node, r can be replicated via main memory copies which is much faster (see step (c)). Thus step (ii) avoids a lot of possible network traffic.

As soon as a hash buffer is full its contents is transmitted to the corresponding output buffer.

(c) A further replication step is performed when a tuple r with a timestamp $[t_s, t_e]$ arrives at an output buffer $\mathcal{O}'_k$ or at a $\mathcal{O}''_k$ with $k \in K_{\mathit{first}}$. Such a buffer is located at processor $j = \mathit{processor}(k)$ which itself resides at node $i = \mathit{node}(\mathit{processor}(k))$. From there, r is replicated within node i and put into the output buffers $\mathcal{O}''_l$ with $l \in K''_P(r)$. This node-internal replication can be done within main memory which is much faster than if it had been performed over the interconnect (see step (ii) in (b)).

[5]It is for simplicity reasons that we assume that these buffers keep tuples. Alternatively, they might hold references to tuples or one could create $2m$ index structures for describing the fragments. This would require very detailed performance modeling without providing any benefit for our purposes. Therefore we assume that the fragments are to be materialised.

Each processor that holds an output buffer as described above replicates r in the following way:

/* $k = \mathit{fragment}_P(t_s)$ or $k \in K_{\mathit{first}}$ */

for $l = k + 1$ **to** $\min\{\max K''_P(r), \mathit{last\text{-}node}(i)\}$ **do**
    send tuple to the output buffer $\mathcal{O}'_l$
od

(d) When an output buffer is full then its tuples are flushed to disk.

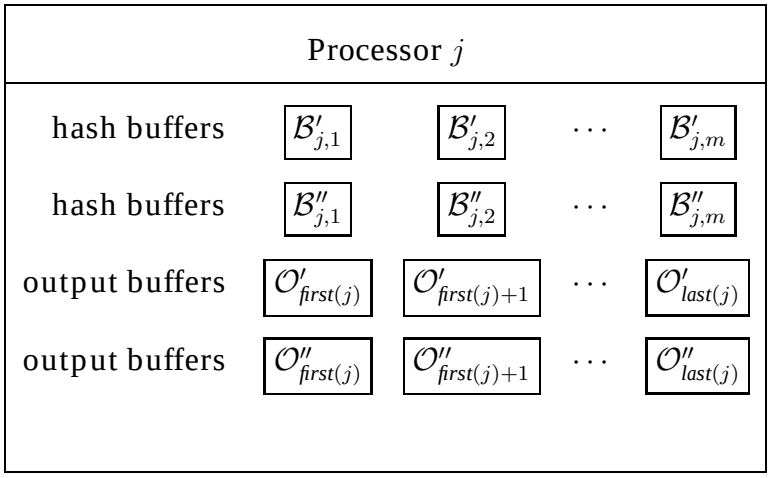


Figure 8.11: Buffers at processor j.

The significant difference, in comparison to partitioning for a traditional parallel join, is the replication of tuples in steps (b).(ii) and (c). We chose a two-level replication: (b).(ii) replicates the tuples over the interconnect and positions the tuples on all nodes that have a processor that has to process the respective tuple. This step can be regarded as an *inter-node replication*. Step (c) replicates the tuples within the nodes and sends them to the remaining processors. This *intra-node replication* is faster because it can be done via shared-memory rather than via communication over the interconnection network. If this step was incorporated into step (b).(ii) the advantage of fast communication via main memory would be lost.

As already stated, there are more efficient ways to repartition R and Q, e.g. by building index structures to represent the new fragments rather than materialising them as described above. We will later see that the repartitioning stage, even when performed in this non-optimal manner, still contributes only

a minor part to the overall costs. It is the joining stage that dominates the overall performance. For this reason we chose the naive approach for repartitioning which not only improves the readability but also simplifies the cost model that is created in section 8.4.

8.3.4 Stage 2: Joining

We now look at stage 2 of the algorithm. Here, each processor performs one or more computations RQ_k with $k = 1, \ldots, m$ in sequential order (see figure 8.10). In the remainder, we focus on one single computation RQ_k only and describe how it is done.

(8.1) defined an RQ_k to consist of the sequential computation of three individual *subjoins*

$$R'_k \bowtie_C Q''_k, \quad R'_k \bowtie_C Q'_k, \quad R''_k \bowtie_C Q'_k$$

Splitting the one 'big' join $R \bowtie_C Q$ into several smaller partial joins $R_1 \bowtie_C Q_1$, $\ldots$, $R_m \bowtie_C Q_m$ and each of these partial joins into subjoins as in (8.1) was described in section 4.4.3. The two major advantages for this are:

1. It is

   $$R_k \bowtie_C Q_k \;=\; R'_k \bowtie_C Q'_k \cup R'_k \bowtie_C Q''_k \cup R''_k \bowtie_C Q'_k \cup R''_k \bowtie_C Q''_k$$

   but the join $R''_k \bowtie_C Q''_k$ is redundant as it is

   $$R''_k \bowtie_C Q''_k \;\subseteq\; R_{k-1} \bowtie_C Q_{k-1}$$

   Processing can therefore be reduced to the first three subjoins, thus avoiding a considerable amount of unnecessary computation.

2. A second advantage arises from the fact that the size of the R'_k are easier to control than those of the R_k or R''_k because every tuple appears only in one R'_k but possibly in several R_k or R''_k:

   $$\sum_{k=1}^{m} |R'_k| \;=\; |R|$$
   $$\sum_{k=1}^{m} |Q'_k| \;=\; |Q|$$

   but

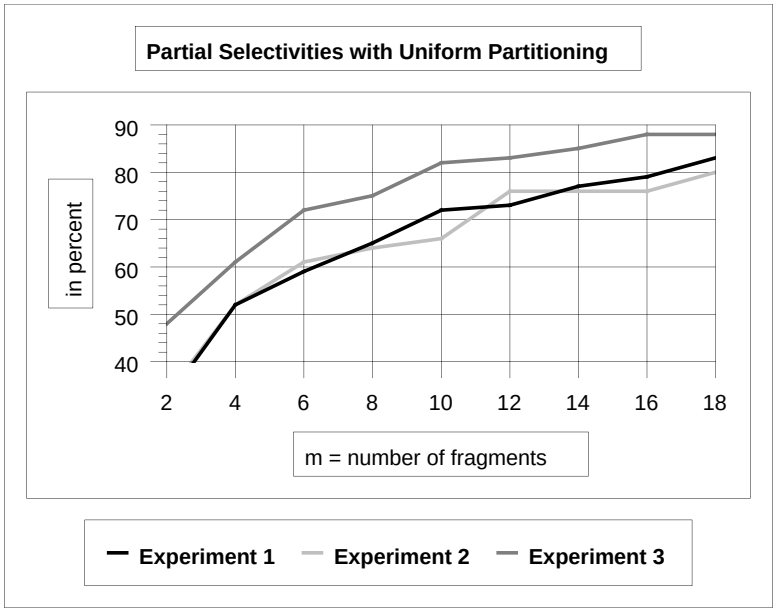
   $$\sum_{k=1}^{m} |R_k| \;\geq\; |R|$$
   $$\sum_{k=1}^{m} |Q_k| \;\geq\; |Q|$$

Actually, it is possible to guarantee that for all $k = 1, \ldots, m$ the R'_k have a certain maximum number X of tuples (see chapter 5 and the oncoming chapter 9). Thus a partition can be chosen that guarantees that all R'_k fit into a processor's local main memory. But this means that each R'_k needs to be read from disk only once for the first subjoin $R'_k \bowtie_C Q''_k$ and is then kept in main memory for the second subjoin $R'_k \bowtie_C Q'_k$. Alternatively one could do the same with Q'_k, computing the subjoin $R''_k \bowtie_C Q'_k$ first, keeping Q'_k in main memory and then computing $R'_k \bowtie_C Q'_k$. Thus we are able to reduce the total number of disk accesses by $|R|/2$ and $|Q|/2$ respectively.

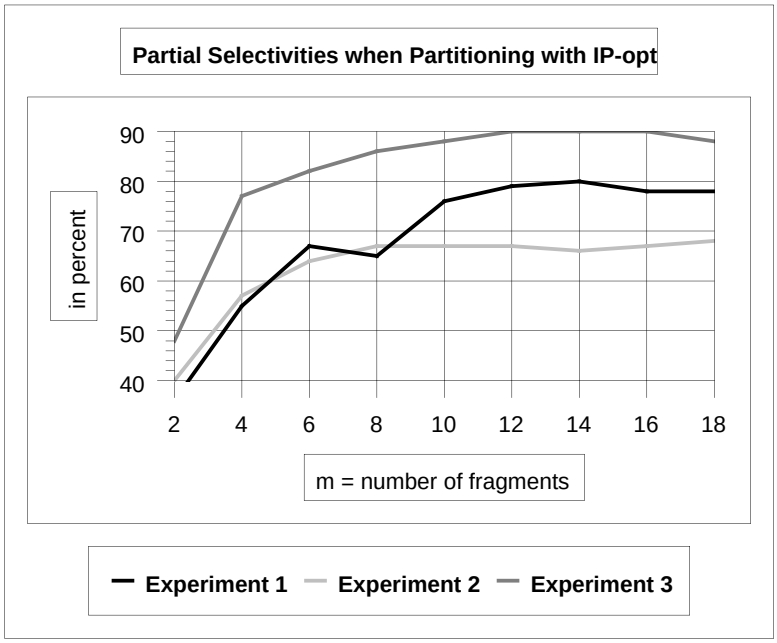
With respect to each subjoin, one can use any sequential temporal join algorithm. We adopt a nested-loops approach for the following reason: the selectivity factors of the partial joins $R_k \bowtie_C Q_k$ – in the remainder we will refer to these factors as *partial selectivities* – can be expected to be fairly high because the data has been partitioned according to the join predicate (i.e. temporal intersection in our case). We performed preliminary experiments in order to confirm this conclusion: two temporal relations U and V of 10000 tuples each were generated. U had a random profile with the majority of tuples being valid in the first half of the lifespan (1440 chronons). V was periodic in the sense that there were several equal peak-to-peak distances for the function giving the number of tuples being valid at a time. The average length of a time interval was the same in both relations (120 chronons). The three temporal intersection joins $U \bowtie_C V$ (experiment 1), $U \bowtie_C U$ (experiment 2) and $V \bowtie_C V$ (experiment 3) were partitioned into m partial joins using (a) a uniform partition of the timeline and (b) an optimal partition using the algorithm `IP-opt` of chapter 5. The resulting partial selectivities are shown in figure 8.12 and confirm our initial conclusion that partial selectivities are fairly high, beyond 70% in most cases. This justifies to use a nested-loops approach. Further partitioning through sorting (sort-merge join) or hashing (hash join) would not increase the performance by much but would introduce an overhead through sorting[6] and hashing respectively.

After having clarified essential details we can now describe how a computation RQ_k is performed. We assume that the subjoins are computed in the

[6]One could argue that the relation might originally be sorted. However, it is difficult to maintain such a sort-order during repartitioning, especially when communication over the interconnect is involved. Most protocols do not guarantee that messages sent in a certain order also arrive in this order. One would then need to restore this order on arrival of the messages. Thus the usage of a sort-merge join algorithm would impose an overhead under any circumstances.



(a)



(b)

Figure 8.12: Partial selectivities as achieved in preliminary experiments.

```
procedure intersection-join (R,Q)
begin
    for each block 𝓡 of R do
        load tuple of 𝓡 into main memory
        for each block 𝓠 of Q do
            load tuple of 𝓠 into main memory
            now compare each r ∈ 𝓡 with each q ∈ 𝓠:
            if [r.t_s, r.t_e] intersects [q.t_s, q.t_e] then
                time-concatenate r and q
                place result in an output buffer O_j
                if O_j is full then
                    flush contents of O_j to disk
                fi
            fi
        od
    od
end
```

Remark: j is the number of the processor on which the intersection join is executed.

Figure 8.13: The procedure *intersection-join(R,Q)*.

order in which they appear in (8.1). Furthermore it is assumed that the respective outer relation is bigger than the corresponding inner relation for efficiency reasons (see section 3.4.1). If this is not the case one can easily swap. The join condition C consists of a temporal intersection and some boolean expression $\mathcal{C}(r,q)$ over tuples $r \in R$ and $q \in Q$. The latter is supposed to be non-temporal and therefore amenable to the same optimisations that may be applied to non-temporal join evaluation. For performance modeling purposes we later assume that $\mathcal{C}(r,q)$ evaluates to **true** so that we can neglect any implications given by this part of the join condition and concentrate on the essential temporal aspects. Finally, a processor j is supposed to accumulate join results in an output buffer $\mathcal{O}_j$ which is flushed to disk when it is full. A join is then performed as shown in figure 8.13 which already assumes that $\mathcal{C}(r,q)$ evaluates to **true**. A computation RQ_k processes the three subjoins in sequential order:

(a) intersection-join(R'_k,Q''_k)

(b) intersection-join(R'_k,Q'_k)

(c) intersection-join(R''_k,Q'_k)

The term *time-concatenate* refers to the process of creating an appropriate timestamp when concatenating a tuple $r \in R$ and a tuple $q \in Q$. This was described

by (4.1) in section 4.1.

8.4 Cost Model

8.4.1 The Basic Issues

The cost model that corresponds to the temporal join processing model of section 8.3 is created in a similar approach as the one taken by Hua *et al.* when they model the performance of a parallel hash join in [Hua et al., 1991]. Hua et al. have shown that with their approach they were able to derive many interesting characteristics and simulate many developments that arose in real-world applications. We have already seen one example in figure 8.6. Another is the convergence towards hybrid parallel architectures – a development that was still neglected in 1992 by DeWitt and Gray in [DeWitt and Gray, 1992] but which has become reality nowadays (see discussion in [Norman et al., 1996]). We can therefore expect equally viable results when following their approach.

Thus we assume the hybrid architecture of section 8.2.3 and expect the temporal join to be processed as described in section 8.3. The costs are measured in seconds. The total response time C_{total} of the temporal join depends on the times C_{part} and C_{join} spent in stages 1 and 2. In reality there might be an overlap between these two stages; thus

$$\max\{C_{part}, C_{join}\} \;\leq\; C_{total} \;\leq\; C_{part} + C_{join}$$

In our model, however, we assume that there is no overlap (e.g. enforced through a barrier type synchronisation). Thus we use the upper bound

$$C_{total} \;=\; C_{part} + C_{join}$$

The stages (a), (b) etc. within stages 1 and 2 are treated accordingly, i.e.[7]

$$\begin{aligned} C_{part} &= C_{1(a)}(R) + C_{1(b)}(R) + C_{1(c)}(R) + C_{1(d)}(R) + \\ &\quad\; C_{1(a)}(Q) + C_{1(b)}(Q) + C_{1(c)}(Q) + C_{1(d)}(Q) \\ C_{join} &= C_{2(a)} + C_{2(b)} + C_{2(c)} \end{aligned}$$

Furthermore we assume that the overlap between the I/O, communication, CPU and memory access phases within each stage is perfect. In reality, this

[7] We note that repartitioning applies to all relations that participate in the join, i.e. R and Q in the prototypical case. Therefore the cost components $C_{1(a)}$, $C_{1(b)}$, $C_{1(c)}$, $C_{1(d)}$ are computed for R using R's parameters – this is indicated by $C_{1(a)}(R)$, $C_{1(b)}(R)$ etc. – and for Q using Q's parameters – this is indicated by $C_{1(a)}(Q)$, $C_{1(b)}(Q)$ etc.

can almost be achieved by separate I/O and communication processors. This means that we have to analyse the costs for I/O, communication, CPU and memory accesses for each substage of stages 1 and 2 and assume the maximum of these partial costs to be relevant for that substage. For stage 1 (a), for example, this means that

$$C_{1(a)} = \max\{C_{1(a),io}, C_{1(a),com}, C_{1(a),cpu}, C_{1(a),mem}\}$$

Section 8.4.2 describes how C_{part} is calculated; section 8.4.3 does the same for C_{join}. This analysis assumes certain parameters like I/O bandwidth, processor speed, amount of memory etc. These are introduced within those sections. As a convention we will use

- i to refer to node indices, i.e. $i \in \{1, \ldots, \mathcal{M}\}$,
- j to refer to processor indices, i.e. $j \in \{1, \ldots, \mathcal{MN}\}$, and
- k to refer to fragment indices, i.e. $k \in \{1, \ldots, m\}$.

If not specified otherwise we will assume the respective nominated range of values for i, j and k.

8.4.2 Stage 1: Repartitioning

The stage – as described in section 8.3.3 – comprises disk accesses, communication, CPU time and memory accesses as the major cost factors. These costs arise for the repartitioning of all participating relations. We restrict ourselves to deriving the cost of the substages for one relation R; the other relations are treated similarly:

(a) **Loading fragments of R from disk**

This substage does not involve any communication or memory accesses. Only disk accesses and the CPU costs for initiating these accesses have to be modelled. We note that we will not distinguish between random and sequential disk I/O because we can expect an equal mix of random and sequential accesses when comparing the costs caused by the various partitioning strategies. This goes along the argument that was mentioned at the end of section 8.1, i.e. that we are interested in a relative comparison rather than absolute figures. However, we stress that a distinction between random and sequential accesses would be more realistic and has therefore been frequently made in the literature, e.g. in [Soo et al., 1994].

We assume a uniform distribution of R over the nodes, i.e. the fragments $\hat{R}_i$ are equally sized. They are stored on the disks of the nodes. Therefore each node i has to move

$$|\hat{R}_i| \approx \frac{|R|}{\mathcal{M}}$$

tuples each of size $|r|$ to its main memory. The disk I/O bandwidth is u_{io} which leads to

$$C_{1(a),io} = \frac{|R|}{\mathcal{M}} \cdot \frac{|r|}{w_{io}} \tag{8.10}$$

as the disk access costs whereby a portion of

$$\frac{|R|}{\mathcal{M}\mathcal{N}}$$

is loaded by each processor. We assume that tuples are moved blockwise[8] from disk to the processors. Thus a disk I/O has to be initiated only once per block (page). If b is the size of such a page then

$$\frac{|R|}{\mathcal{M}\mathcal{N}} \cdot \frac{|r|}{b}$$

is the number of pages to be moved. The time spent on initiating one page movement is

$$\frac{I_{io}}{\mu}$$

where I_{io} is the number of microprocessor instructions necessary; μ is the number of instructions per second being performed by the processor. Thus

$$C_{1(a),cpu} = \frac{|R|}{\mathcal{M}\mathcal{N}} \cdot \frac{|r|}{b} \cdot \frac{I_{io}}{\mu} \tag{8.11}$$

is the CPU time spent in substage 1 (a).

As mentioned in the previous section, we assume that disk I/O, communication, CPU and memory access phases have a perfect overlap. Therefore it is the maximum of the individual times that is finally relevant. The total time spent on substage 1 (a) is therefore the maximum of (8.10) and (8.11):

$$C_{1(a)} = \max\{C_{1(a),io}, C_{1(a),cpu}\} \tag{8.12}$$

[8] or – in other terms – pagewise.

(b) **Redistribution of the data via the network, including inter-node replication**

Substage 1 (b) describes the distribution of the data between the nodes. It hashes tuples to the hash buffers and initiates an inter-node replication via the interconnect. Thus it comprises communication, CPU and memory costs.

Each of the $\mathcal{M}$ nodes has to deal with

$$\frac{|R|}{\mathcal{M}}$$

tuples. These are distributed over the interconnect to other nodes depending on their respective timestamp. We assume that $1/\mathcal{M}$-th of the tuples are not sent over the interconnect but remain at the node, i.e. a share of

$$\frac{\mathcal{M}-1}{\mathcal{M}}$$

of the tuples is actually involved in inter-node communication. The data comprises not only the primary but also the replicated tuples. We use the parameter δ_R to denote the average number of times that a tuple of R is replicated on the basis of an underlying partition P:

$$\begin{aligned}
\delta_R &= \frac{1}{|R|} \cdot \left(\sum_{k=1}^{m} |R'_k| + |R''_k| \right) \\
&= \frac{1}{|R|} \cdot \left(\sum_{k=1}^{m} |R'_k| + \sum_{k=1}^{m} |R''_k| \right) \\
&= \frac{1}{|R|} \cdot \left(|R| + \sum_{k=1}^{m} |R''_k| \right) \\
&= 1 + \frac{1}{|R|} \sum_{k=1}^{m} |R''_k| \qquad (8.13)
\end{aligned}$$

However, communication via the interconnect is only necessary for inter-node replication, i.e. replication across node boundaries. A node boundary appears every $\alpha\mathcal{N}$-th fragment R_k in the beginning, later every $\beta\mathcal{N}$-th fragment. On average this happens every $\mathcal{N}$-th fragment if $m \leq \mathcal{M}\mathcal{N}$ or every $m/\mathcal{M}$ fragments otherwise. This translates to node boundaries being encountered every $\max\{\mathcal{N}, m/\mathcal{M}\}$ fragments on average. Thus, on average, a tuple is replicated over node boundaries

$$\gamma_R = \frac{\delta_R}{\max\{\mathcal{N}, m/\mathcal{M}\}} \qquad (8.14)$$

times. In other words: whereas δ_R provides the average number of fragments R_k in which a tuple has to be present, γ_R gives the average number of nodes over which these R_k are spread. Hence each node sends

$$\frac{\mathcal{M}-1}{\mathcal{M}} \cdot \frac{|R|}{\mathcal{M}} \cdot \gamma_R \cdot |r|$$

bytes. As each of the $\mathcal{M}$ node sends this amount the total communication costs are

$$C_{1(b),com} \;=\; \frac{\mathcal{M}-1}{\mathcal{M}} \cdot |R| \cdot \gamma_R \cdot \frac{|r|}{w_{com}} \tag{8.15}$$

where w_{com} refers to the communication bandwidth. To initiate the communication for a tuple transfer each processor is supposed to perform I_{com} instructions, thus it has to spend

$$\frac{I_{com}}{\mu}$$

seconds per transfer. Similarly, the CPU costs for hashing a tuple to a buffer are

$$\frac{I_{hash}}{\mu}$$

if I_{hash} is the average number of CPU instructions for the hashing. For all r hashed by a single processor there arise CPU costs of

$$\frac{|R|}{\mathcal{M}\mathcal{N}} \cdot \frac{I_{hash}}{\mu}$$

The total CPU costs are therefore

$$C_{1(b),cpu} \;=\; \frac{\mathcal{M}-1}{\mathcal{M}} \cdot \frac{|R|}{\mathcal{M}\mathcal{N}} \cdot \gamma_R \cdot \frac{I_{com}}{\mu} \;+\; \frac{|R|}{\mathcal{M}\mathcal{N}} \cdot \frac{I_{hash}}{\mu} \tag{8.16}$$

Finally, on each node there are

$$\frac{|R|}{\mathcal{M}} \cdot \delta_R$$

tuples to be moved to buffers in main memory. This causes memory access costs of

$$\frac{|r|}{w_{mem}}$$

per tuples with w_{mem} referring to the bandwidth for main memory accesses. Thus

$$C_{1(b),mem} \;=\; \frac{|R|}{\mathcal{M}} \cdot \delta_R \cdot \frac{|r|}{w_{mem}} \tag{8.17}$$

are the total costs for memory accesses per node. The total time spent on stage 1 (b) is the maximum of (8.15), (8.16) and (8.17):

$$C_{1(b)} \;=\; \max\{C_{1(b),com}, C_{1(b),cpu}, C_{1(b),mem}\} \tag{8.18}$$

(c) **Intra-node replication via main memory**

Stage 1 (c) replicates tuples within a node. This can be done via main memory. Originally, a node i had to cope with

$$|\hat{R}_i| \approx \frac{|R|}{\mathcal{M}}$$

tuples per node. Each tuple is replicated δ_R times on average. If δ_R exceeds $m/\mathcal{M}$ (the average number of fragments per node) then most tuples are replicated over all processors of a node; otherwise just δ_R times. Writing one tuple to memory creates costs of

$$\frac{|r|}{w_{mem}}$$

if w_{mem} is the memory bandwidth in bytes per second. Thus the memory access costs for this stage are

$$C_{1(c),mem} = \frac{|R|}{\mathcal{M}} \cdot \frac{|r|}{w_{mem}} \cdot \min\{\delta_R, \frac{m}{\mathcal{M}}\} \tag{8.19}$$

CPU costs for these memory accesses can be neglected as they comprise by far less instructions as computing expressions (e.g. as I_{hash} for hashing) or processing two tuples (see I_{proc}). Thus (8.19) states the total costs for stage 1 (c):

$$C_{1(c)} = C_{1(c),mem} \tag{8.20}$$

(d) **Writing new fragments of R to disk**

Stage 1 (d) writes the new fragments R'_1, R''_1, R'_2, R''_2, ... , R'_m, R''_m to disk. A node i has to move

$$\sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} |R'_k| + |R''_k|$$

tuples to its disks which results in I / O costs of

$$\left(\sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} |R'_k| + |R''_k| \right) \cdot \frac{|r|}{w_{io}}$$

However, as this is performed concurrently it is only the costs of the node that takes longest to perform the task that are relevant. Thus the I / O costs for this stage are

$$C_{1(d),io} = \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} |R'_k| + |R''_k| \right\} \cdot \frac{|r|}{w_{io}} \tag{8.21}$$

By analogy to stage 1 (a), this I/O has to be initiated by the CPUs with each page movement requiring I_{io} instructions. A processor j moves

$$\sum_{k=first(j)}^{last(j)} |R'_k| + |R''_k|$$

tuples to the disks of its node. Again, for the overall costs only the processor that has the highest workload is relevant. This leads to overall CPU costs of

$$C_{1(d),cpu} = \max_{j=1}^{\mathcal{MN}} \left\{ \sum_{k=first(j)}^{last(j)} |R'_k| + |R''_k| \right\} \cdot \frac{|r|}{b} \cdot \frac{I_{io}}{\mu} \tag{8.22}$$

The total costs for stage 1 (d) arise from the maximum of (8.21) and (8.22):

$$C_{1(d)} = \max\{C_{1(d),io}, C_{1(d),cpu}\} \tag{8.23}$$

The cost components of stage 1 are summarised in tables 8.1, 8.2, 8.3 and 8.4. Relation Q has to be repartitioned in the same way using the respective parameters $|Q|$, $|q|$, δ_Q etc. With equations (8.12), (8.18), (8.20) and (8.23) we can derive the total costs of stage 1 as

$$\begin{aligned} C_{part} = {} & C_{1(a)}(R) + C_{1(b)}(R) + C_{1(c)}(R) + C_{1(d)}(R) + \\ & C_{1(a)}(Q) + C_{1(b)}(Q) + C_{1(c)}(Q) + C_{1(d)}(Q) \end{aligned} \tag{8.24}$$

with $C_{1(a)}(R)$ indicating that parameters of relation R should be used for computing the costs of stage 1 (a). Similarly, it has to be distinguished between R and Q in the other substages.

8.4.3 Stage 2: Joining

In this section we derive the costs C_{join} of stage 2 of the temporal join processing model. This stage is described in section 8.3.4. Here, each processor j performs computations $RQ_{first(j)}$, $RQ_{first(j)+1}$, ... , $RQ_{last(j)}$. Each computation RQ_k consists of performing the three subjoins

$$\begin{aligned} &(a) \quad R'_k \bowtie_C Q''_k \\ &(b) \quad R'_k \bowtie_C Q'_k \\ &(c) \quad R''_k \bowtie_C Q'_k \end{aligned}$$

Stage 1 (a)	
Disk I/O	$\frac{\lvert R\rvert}{\mathcal{M}} \cdot \frac{\lvert r\rvert}{w_{io}}$
Communication	
CPU	$\frac{\lvert R\rvert}{\mathcal{MN}} \cdot \frac{\lvert r\rvert}{b} \cdot \frac{I_{io}}{\mu}$
Memory	

Table 8.1: Cost components for stage 1 (a).

Stage 1 (b)	
Disk I/O	
Communication	$\frac{\mathcal{M}-1}{\mathcal{M}} \cdot \lvert R\rvert \cdot \gamma_R \cdot \frac{\lvert r\rvert}{w_{com}}$
CPU	$\frac{\mathcal{M}-1}{\mathcal{M}} \cdot \frac{\lvert R\rvert}{\mathcal{MN}} \cdot \gamma_R \cdot \frac{I_{com}}{\mu} + \frac{\lvert R\rvert}{\mathcal{MN}} \cdot \frac{I_{hash}}{\mu}$
Memory	$\frac{\lvert R\rvert}{\mathcal{M}} \cdot \delta_R \cdot \frac{\lvert r\rvert}{w_{mem}}$

Table 8.2: Cost components for stage 1 (b).

Stage 1 (c)	
Disk I/O	
Communication	
CPU	
Memory	$\frac{\lvert R\rvert}{\mathcal{M}} \cdot \frac{\lvert r\rvert}{w_{mem}} \cdot \min\{\delta_R, \frac{m}{\mathcal{M}}\}$

Table 8.3: Cost components for stage 1 (c).

Stage 1 (d)	
Disk I/O	$\max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R'_k \rvert + \lvert R''_k \rvert \right\} \cdot \frac{\lvert r \rvert}{w_{io}}$
Communication	
CPU	$\max_{j=1}^{\mathcal{MN}} \left\{ \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R'_k \rvert + \lvert R''_k \rvert \right\} \cdot \frac{\lvert r \rvert}{b} \cdot \frac{I_{io}}{\mu}$
Memory	

Table 8.4: Cost components for stage 1 (d).

in exactly that order in order to exploit the opportunity to keep (a part of) R'_k in main memory at the end of join (a), thus avoiding to reload it at the beginning of join (b). Alternatively, we could use the order (c), (b), (a) and exploit the same fact for the Q'_k. In the remainder, however, we will assume the order (a), (b), (c); arguments and results can be easily transferred to the alternative case.

The subjoins are performed by using the nested-block join technique. This means that in joins (a) and (b) an R'_k is cached either entirely or blockwise in main memory while being joined with Q''_k and Q'_k respectively. In join (c) we assume the same for the Q'_k when being joined with R''_k. The number of blocks in which an R'_k has to be divided is referred to by λ_k. If *mem* is the amount of main memory at each node then each processor gets a share of

$$\frac{\textit{mem}}{\mathcal{N}}$$

Thus λ_k can be computed by

$$\lambda_k \;=\; \left\lceil \lvert R'_k \rvert \cdot \lvert r \rvert \cdot \frac{\mathcal{N}}{\textit{mem}} \right\rceil$$

Similarly, φ_k refers to the number of blocks into which a Q'_k is divided for computation in subjoin (c):

$$\varphi_k \;=\; \left\lceil \lvert Q'_k \rvert \cdot \lvert q \rvert \cdot \frac{\mathcal{N}}{\textit{mem}} \right\rceil$$

We note that λ_k and φ_k are 1 if R'_k and Q'_k respectively fit in main memory.

The costs for each of the subjoins are computed in the same way: the two sets of tuples have to be read from disk and are then joined using a nested-blocked approach[9]. A minor difference appears for join (b) that can exploit the

[9]See section 3.4.1.

fact that (a block of) R'_k already resides in main memory and therefore has not to be reloaded from disk.

There is no communication via the interconnect involved because the computations RQ_k are independent from each other. In the following paragraphs, we describe the cost components for join (a). The costs for joins (b) and (c) are derived accordingly with the marginal difference in the case of join (b) that has been mentioned above. All cost components are summarised in tables 8.5, 8.6 and 8.7.

First, we look at the I/O costs. These have to be considered on a node-wide level as disks are shared among all the processors of a node. For join (a), all tuples of R'_k have to be read once from disk and those of Q''_k once per block of R'_k, i.e. λ_k times. Thus a node i has to load

$$\sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} |R'_k| + |Q''_k| \cdot \lambda_k$$

tuples of sizes $|r|$ and $|q|$ respectively. This implies I/O costs of

$$\frac{1}{w_{io}} \cdot \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} |R'_k| \cdot |r| \; + \; |Q''_k| \cdot |q| \cdot \lambda_k$$

at node i. For the overall costs only the costs of the node with the heaviest load is relevant, i.e.

$$C_{2(a),io} \; = \; \frac{1}{w_{io}} \cdot \max_{i=1}^{M} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} |R'_k| \cdot |r| \; + \; |Q''_k| \cdot |q| \cdot \lambda_k \right\} \qquad (8.25)$$

As in stage 1, the individual processors have to initiate the I/O transfers. I_{io} CPU instructions are required per page transfer. Furthermore, every tuple in R'_k is tested with every tuple in Q''_k which makes

$$\sum_{k=\textit{first}(j)}^{\textit{last}(j)} |R'_k| \cdot |Q''_k|$$

tests on a processor j. We assume that processing a pair of tuples requires I_{proc} instructions. This means that the CPU costs for a processor j are

$$\frac{1}{b} \cdot \frac{I_{io}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} |R'_k| \cdot |r| \; + \; |Q''_k| \cdot |q| \cdot \lambda_k \quad + \quad \frac{I_{proc}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} |R'_k| \cdot |Q''_k|$$

For the overall costs, only the processor with the heaviest load is relevant. Therefore the CPU costs for join (a) are

$$C_{2(a),cpu} = \max_{j=1}^{\mathcal{MN}} \left\{ \frac{1}{b} \cdot \frac{I_{io}}{\mu} \cdot \sum_{k=first(j)}^{last(j)} |R'_k| \cdot |r| + |Q''_k| \cdot |q| \cdot \lambda_k + \frac{I_{proc}}{\mu} \cdot \sum_{k=first(j)}^{last(j)} |R'_k| \cdot |Q''_k| \right\} \quad (8.26)$$

The joining stage requires one main memory access to a tuple $r \in R'_k$ per tuple $q \in Q''_k$, i.e.

$$|R'_k| \cdots |Q''_k|$$

accesses in total, each retrieving a tuple of size $|r|$. Memory – as disks – is shared among all processors per node. Costs for main memory access therefore have to be considered on a node-wide level. Thus on a node i there are

$$\sum_{k=first\text{-}node(i)}^{last\text{-}node(i)} |R'_k| \cdot |Q''_k|$$

accesses to a tuple of size $|r|$ which produces costs of

$$\frac{|r|}{w_{mem}} \cdot \sum_{k=first\text{-}node(i)}^{last\text{-}node(i)} |R'_k| \cdot |Q''_k|$$

Again, for the overall costs only the costs of the node with the heaviest load is relevant, i.e.

$$C_{2(a),mem} = \frac{|r|}{w_{mem}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=first\text{-}node(i)}^{last\text{-}node(i)} |R'_k| \cdot |Q''_k| \right\} \quad (8.27)$$

The total costs for stage 2 (a) – the join (a) – arise from the maximum of equations (8.25), (8.26) and (8.27):

$$C_{2(a)} = \max\{C_{2(a),io}, C_{2(a),cpu}, C_{2(a),mem}\} \quad (8.28)$$

As explained above, the costs $C_{2(b)}$ for join (b) and $C_{2(c)}$ for join (c) are similarly derived. Tables 8.5, 8.6 and 8.7 summarise the cost components for stage 2. The total costs C_{join} for this stage are

$$C_{join} = C_{2(a)} + C_{2(b)} + C_{2(c)} \quad (8.29)$$

Stage 2 (a)	
Disk I/O	$\frac{1}{w_{io}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R'_k \rvert \cdot \lvert r \rvert \; + \; \lvert Q''_k \rvert \cdot \lvert q \rvert \cdot \lambda_k \right\}$
CPU	$\max_{j=1}^{\mathcal{MN}} \left\{ \frac{1}{b} \cdot \frac{I_{io}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R'_k \rvert \cdot \lvert r \rvert + \lvert Q''_k \rvert \cdot \lvert q \rvert \cdot \lambda_k + \frac{I_{proc}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R'_k \rvert \cdot \lvert Q''_k \rvert \right\}$
Memory	$\frac{\lvert r \rvert}{w_{mem}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R'_k \rvert \cdot \lvert Q''_k \rvert \right\}$

Table 8.5: Cost components for the joining stage 2 (a).

Stage 2 (b)	
Disk I/O	$\frac{1}{w_{io}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R'_k \rvert \cdot \lvert r \rvert \cdot \frac{\lambda_k - 1}{\lambda_k} \; + \; \lvert Q'_k \rvert \cdot \lvert q \rvert \cdot \lambda_k \right\}$
CPU	$\max_{j=1}^{\mathcal{MN}} \left\{ \frac{1}{b} \cdot \frac{I_{io}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R'_k \rvert \cdot \lvert r \rvert \cdot \frac{\lambda_k - 1}{\lambda_k} \; + \; \lvert Q'_k \rvert \cdot \lvert q \rvert \cdot \lambda_k + \frac{I_{proc}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R'_k \rvert \cdot \lvert Q'_k \rvert \right\}$
Memory	$\frac{\lvert r \rvert}{w_{mem}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R'_k \rvert \cdot \lvert Q'_k \rvert \right\}$

Table 8.6: Cost components for the joining stage 2 (b).

Stage 2 (c)	
Disk I/O	$\frac{1}{w_{io}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R''_k \rvert \cdot \lvert r \rvert \cdot \varphi_k \; + \; \lvert Q'_k \rvert \cdot \lvert q \rvert \right\}$
CPU	$\max_{j=1}^{\mathcal{MN}} \left\{ \frac{1}{b} \cdot \frac{I_{io}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R''_k \rvert \cdot \lvert r \rvert \cdot \varphi_k \; + \; \lvert Q'_k \rvert \cdot \lvert q \rvert + \frac{I_{proc}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R''_k \rvert \cdot \lvert Q'_k \rvert \right\}$
Memory	$\frac{\lvert q \rvert}{w_{mem}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R''_k \rvert \cdot \lvert Q'_k \rvert \right\}$

Table 8.7: Cost components for the joining stage 2 (c).

8.5 Evaluation of Characteristics

Before designing techniques for partitioning temporal data we want to evaluate some characteristics of the performance model that was described in sections 8.2 – 8.4. Interesting questions, for example, are:

- Which cost components dominate? Is it I/O, communication, CPU time or memory accesses?
- What is the ratio between C_{part} and C_{join} ?
- Which workload parameters affect which cost components?

The problem about answering these questions is that the partition that underlies the simulated join computation is an important factor within the cost model. Therefore it is impossible to evaluate the entire performance model without making any assumptions about the underlying partition. In order to reduce the importance of that we will create experiments for computing temporal intersection joins

$$R \bowtie_C Q \;=\; R_1 \bowtie_C Q_1 \cup \cdots \cup R_m \bowtie_C Q_m$$

assuming that the underlying data is uniform.

Section 8.5.1 describes the uniform workload. In section 8.5.2, this workload is used to perform a series of experiments that provide useful information about the characteristics of the performance model.

8.5.1 Uniform Workloads

In practice, uniform workloads are a very rare exception. Therefore they cannot be used to draw realistic conclusions. However, they can help to simplify the evaluation of the components of the performance model that are not susceptible to data skew and other forms of non-uniformity. To prepare such a preliminary evaluation is the focus of this section. Under the notion of uniformity of the workload we assume the following:

- All intervals in R and Q have the same length τ.
- The startpoints of these intervals are uniformly distributed over the timeline, i.e. at any time $t \in L(R \cup Q)$ there is the same number of intervals starting as at a time $t' \in L(R \cup Q)$ with $t' \neq t$.

- The only input parameter to define a partition is m. The $(m-1)$ breakpoints are supposed to be at equal distances.
- Both relations have the same span, i.e.

$$L(R) \;=\; L(Q) \;=\; L(R \cup Q)$$

- The previous points imply that

$$|R_1| = |R_2| = \cdots = |R_m|$$

$$|Q_1| = |Q_2| = \cdots = |Q_m|$$

- We also assume that both relations have the same number of tuples, i.e.

$$|R| \;=\; |Q|$$

- Tuples in R and Q are supposed to be of an equal size, i.e.

$$|r| \;=\; |q|$$

  for $r \in R$ and $q \in Q$.

Following these assumptions on uniform data, we can derive the necessary parameters for the costs model.

First, we want to approximate the size of a fragment R_k with $k \in \{1, \ldots, m\}$. For that purpose we can use δ_R which gives the average number of fragments to which a tuple $r \in R$ is assigned. Assuming a partition of $L(R \cup Q)$ into m equally sized segments, i.e. any two subsequent breakpoints, p_k and p_{k+1}, are at equal distance. The latter can be calculated by $\frac{|L(R \cup Q)|}{m}$. If τ is the average length of a timestamp interval then the interval occupies a portion of

$$\frac{\tau}{\frac{|L(R \cup Q)|}{m}} \tag{8.30}$$

of a segment. As interval startpoints are uniformly distributed over a segment, there are some intervals starting in close enough proximity to the end of a segment to overlap a breakpoint. Therefore we have to add 1 to the value obtained by (8.30) in order to get an estimate for δ_R:

$$\delta_R \;=\; \frac{m}{|L(R \cup Q)|} \cdot \tau + 1$$

Figure 8.14 shows an example for $\tau = 4$, $\frac{|L(R \cup Q)|}{m} = 10$ and one interval in R starting at each point $t \in L(R \cup Q)$. (8.30) gives a value of 0.4 which means that

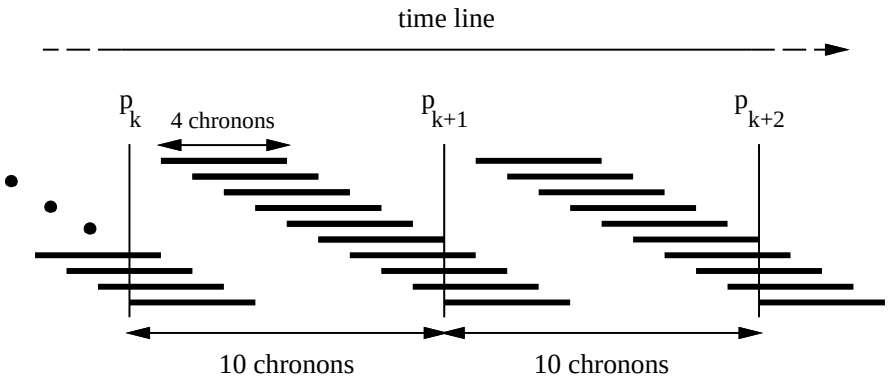


Figure 8.14: An example for the approximation of δ_R for $\frac{|L(R \cup Q)|}{m} = 10$ chronons and $\tau = 4$ chronons.

40% of the intervals starting within a segment overlap the right breakpoint. In the example of figure 8.14, these comprise 4 out of 10 intervals. From a global point of view this implies that 40% of the relation's tuples overlap a breakpoint. Therefore it is

$$\sum_{k=1}^{m} |R_k| \;=\; 1.4 \cdot |R| \;=\; \delta_R \cdot |R|$$

Thus $\delta_R = 1.4$ in this example.

The quotient in (8.30) can also result in values beyond 1 which means that $\frac{|L(R \cup Q)|}{m} < \tau$ which means that the (average) length of a segment is smaller than the average length of an interval. Consequently, most (in the general case) or all (under the assumption of uniformity) intervals overlap the right breakpoint. This is a bad choice for a partition.

Now we can approximate the size of a fragment R_k for $k \in \{1, \ldots, m\}$:

$$|R_k| \;=\; \frac{|R|}{m} \cdot \delta_R$$

As each tuple can only be assigned to one R'_k, the sizes of these are given by

$$|R'_k| \;=\; \frac{|R|}{m}$$

From $|R_k| = |R'_k| + |R''_k|$ we can conclude that

$$|R''_k| \;=\; |R_k| - |R'_k| \;=\; \frac{|R|}{m} \cdot (\delta_R - 1)$$

In the same way, the values for the $|Q_k|$, $|Q'_k|$ and $|Q''_k|$ can be derived. Table 8.8 summarises the approximations made under the assumption of uniformity.

Parameter	Approximation
$\|R_k\|$	$\frac{\|R\|}{m} \cdot \delta_R$
$\|R'_k\|$	$\frac{\|R\|}{m}$
$\|R''_k\|$	$\frac{\|R\|}{m} \cdot (\delta_R - 1)$
δ_R	$\frac{m}{\|L(R \cup Q)\|} \cdot \tau + 1$

(a) Relation R

Parameter	Approximation
$\|Q_k\|$	$\frac{\|Q\|}{m} \cdot \delta_Q$
$\|Q'_k\|$	$\frac{\|Q\|}{m}$
$\|Q''_k\|$	$\frac{\|Q\|}{m} \cdot (\delta_Q - 1)$
δ_Q	$\frac{m}{\|L(R \cup Q)\|} \cdot \tau + 1$

(b) Relation Q

Table 8.8: Summary of the approximations under uniformity.

8.5.2 Experiments

A series of four experiments was conducted in order to explore the contribution of the various cost components to the overall costs under the assumption of a uniform workload. For this purpose, we assumed a parallel architecture comprising $\mathcal{M} \cdot \mathcal{N} = 16$ processors and performance parameters as outlined in table 8.9. We used average values as provided by manufacturers of parallel hardware and commodity products, e.g. as in [Compaq Computer Corp., 1997], [Tandem Computers GmbH, 1997] or [Seagate Technology, 1997]. In each of the experiments a certain parameter of the workload was varied. The following parameter values were used unless the respective parameter was varied in the experiment:

$$\begin{aligned} \mathcal{M} &= 4 \\ \mathcal{N} &= 4 \\ m &= 16 \\ |R| = |Q| &= 120000 \text{ tuples} \\ |r| = |q| &= 500 \text{ bytes} \\ \tau &= 300 \text{ chronons} \\ |L(R \cup Q)| &= 10000 \text{ chronons} \end{aligned}$$

In the first experiment, the dependency of the hardware configuration, i.e. the distribution of the 16 processors between nodes, was investigated. We used $\mathcal{M} = 1, 2, 4, 8, 16$ nodes. The results for the various cost components are shown

in table 8.10 and in figure 8.15. As it became obvious that the repartitioning costs, C_{part}, would only play a minor role in the overall costs we omitted to present the cost component values for the repartitioning stage (see tables 8.1 – 8.4). The only difference between the four configurations is in the memory access costs: the $\mathcal{M} = 1$ and $\mathcal{M} = 2$ node configuration suffer from all processors accessing the same physical memory. This underlines the viability of our performance model as, in practice, the bus becomes frequently the bottleneck in such configurations. This problem gradually goes away with an increasing number of nodes and therefore with memory being more widely distributed.

The second experiment explored the dependency of the cost components on the number m of partial computation into which the join $R \bowtie_C Q$ was divided. The results are shown in table 8.10 and figure 8.16. A general trend is that C_{total} decreases for increasing values of m. However, the best performance results are achieved if m is a multiple of $\mathcal{M} \cdot \mathcal{N} = 16$. Such configurations achieve an optimal match between the number of processors and the number of computations (see figure 8.10). A further significant result is that the CPU and memory access times for the joins $R'_k \bowtie_C Q'_k$ (i.e. $C_{2(b),cpu}$ and $C_{2(b),mem}$) decrease for an increasing m while $C_{2(a),cpu}$, $C_{2(a),mem}$, $C_{2(c),cpu}$ and $C_{2(c),mem}$ remain almost constant and become increasingly dominant for the total costs during this process. This is a significant conclusion as it underlines the necessity and the potential for a minimisation (or at least reduction) of overlapping intervals by choosing an adequate partition.

In the third experiment, the dependency of the costs on the size of the participating relation was analysed. $|R|$ and $|Q|$ were increased by 20000 tuples in each step with an initial number of 20000 tuples. The results are shown in table 8.10 and figure 8.17. As for conventional joins, most components show a quadratic behaviour with linearly increasing relation sizes. This reflects the quadratic time complexity of the nested-loops join algorithm.

Finally, in the fourth experiment, we looked at the dependency of costs with respect to the (average) interval length τ. The results are shown in table 8.10 and figure 8.18. As one can expect, the joins $R'_k \bowtie_C Q'_k$ are not affected; therefore all components of $C_{2(b)}$ remain constant when varying τ. All other cost components show a linear increase with a linearly increasing value of τ.

8.5.3 Conclusions

We summarise the main conclusions that can be drawn form the results that have been described in the previous section:

Parameter	Description	Value
$\mathcal{M} \cdot \mathcal{N}$	total number of processors	16
μ	processor speed in MIPS	200 MIPS
mem	free main memory per node	32 MB
w_{io}	disk I/O bandwidth per node	20 MB/sec
w_{com}	communication bandwidth	40 MB/sec
w_{mem}	memory bandwidth per node	400 MB/sec
I_{proc}	number of CPU instructions for processing a tuple in each step	1000
I_{hash}	number of CPU instructions for hashing a tuple	1000
I_{com}	number of CPU instructions for initiating a data transfer	500
I_{io}	number of CPU instructions for initiating a disk I/O	500
b	page size	4 kB

Table 8.9: The parameters describing the parallel architecture that is used in the experiments.

1. In all experiments C_{join} clearly dominates C_{total}; C_{part} can be ignored. This has two consequences:

   - It proves that the performance model is largely independent of the underlying parallel programming paradigm, such as message-passing or multithreading, as this fact is mainly relevant for the repartitioning stage.
   - When designing partitioning strategies, we can concentrate on optimising C_{join}.

2. The CPU time dominates the costs for the three subjoins. Memory costs can become important in the case of a hardware platform in which a large number of processors shares the access to the common main memory (see experiment 1).

3. The subjoins that involve replicated tuples, i.e. subjoins (a) and (c), become increasingly important

   - for large numbers, m, of fragments (see experiment 2),
   - for large average interval lengths, τ (see experiment 4).

We have to keep these major influences in mind when designing partitioning strategies for non-uniform situations which have to be expected in reality.

	Input Parameters					C_{part}	$C_{2(a)}$			$C_{2(b)}$			$C_{2(c)}$			C_{join}	C_{total}
	$\mathcal{M}$	$\mathcal{N}$	m	$\|R\|, \|Q\|$	τ		I/O	CPU	Mem	I/O	CPU	Mem	I/O	CPU	Mem		
Experiment 1	1	16	16	120000	300	15.0	5.6	135.0	515.0	7.2	281.3	1072.9	5.6	135.0	515.0	2102.9	2117.9
	2	8	16	120000	300	7.6	2.1	135.0	257.5	1.4	281.3	536.4	2.1	135.0	257.5	1051.4	1059.0
	4	4	16	120000	300	4.4	1.1	135.0	128.7	0.7	281.3	268.2	1.1	135.0	128.7	551.3	555.7
	8	2	16	120000	300	3.7	0.5	135.0	64.4	0.4	281.3	134.1	0.5	135.0	64.4	551.3	554.9
	16	1	16	120000	300	4.9	0.3	135.0	32.2	0.2	281.3	67.1	0.3	135.0	32.2	551.3	556.1
Experiment 2	4	4	12	120000	300	4.9	1.3	180.0	171.7	1.0	500.0	476.8	1.3	180.0	171.7	860.0	864.9
	4	4	14	120000	300	4.6	1.2	154.3	147.1	0.8	367.4	350.4	1.2	154.3	147.1	675.9	680.5
	4	4	16	120000	300	4.4	1.1	135.0	128.7	0.7	281.3	268.2	1.1	135.0	128.7	551.3	555.7
	4	4	18	120000	300	5.2	1.5	240.0	171.6	1.0	444.5	317.9	1.5	240.0	171.6	924.4	929.6
	4	4	20	120000	300	5.9	1.8	215.9	205.9	1.1	360.0	343.3	1.8	215.9	205.9	791.9	797.8
	4	4	22	120000	300	5.7	1.7	196.3	187.2	1.0	297.6	283.8	1.7	196.3	187.2	690.2	695.9
	4	4	24	120000	300	5.4	1.6	180.0	171.7	1.0	250.0	238.4	1.6	180.0	171.7	610.0	615.5
	4	4	26	120000	300	5.3	1.6	166.2	158.5	0.9	213.1	203.2	1.6	166.2	158.5	545.4	550.7
	4	4	28	120000	300	5.1	1.5	154.3	147.1	0.8	183.7	175.2	1.5	154.3	147.1	492.2	497.4
	4	4	30	120000	300	5.0	1.4	144.0	137.3	0.8	160.0	152.6	1.4	144.0	137.3	448.0	453.0
	4	4	32	120000	300	4.9	1.4	135.0	128.7	0.7	140.6	134.1	1.4	135.0	128.7	410.6	415.5
Experiment 3	4	4	16	20000	300	0.7	0.2	3.8	3.6	0.1	7.8	7.5	0.2	3.8	3.6	15.3	16.1
	4	4	16	40000	300	1.5	0.4	15.0	14.3	0.2	31.3	29.8	0.4	15.0	14.3	61.3	62.7
	4	4	16	60000	300	2.2	0.5	33.8	32.2	0.4	70.3	67.1	0.5	33.8	32.2	137.8	140.0
	4	4	16	80000	300	3.0	0.7	60.0	57.2	0.5	125.0	119.2	0.7	60.0	57.2	245.0	248.0
	4	4	16	100000	300	3.7	0.9	93.8	89.4	0.6	195.3	186.3	0.9	93.8	89.4	382.8	386.5
Experiment 4	4	4	16	120000	200	4.1	0.9	90.0	85.8	0.7	281.3	268.2	0.9	90.0	85.8	461.2	465.3
	4	4	16	120000	400	4.8	1.2	180.0	171.7	0.7	281.3	268.2	1.2	180.0	171.7	641.3	646.0
	4	4	16	120000	600	5.4	1.4	270.0	257.5	0.7	281.3	268.2	1.4	270.0	257.5	821.3	826.7
	4	4	16	120000	800	6.1	1.6	360.0	343.3	0.7	281.3	268.2	1.6	360.0	343.3	1001.3	1007.3
	4	4	16	120000	1000	6.7	1.9	450.0	429.1	0.7	281.3	268.2	1.9	450.0	429.1	1181.2	1187.9
	4	4	16	120000	1200	7.4	2.1	540.0	515.0	0.7	281.3	268.2	2.1	540.0	515.0	1361.3	1368.6

Cost values are in seconds.

Table 8.10: Results of the four experiments.

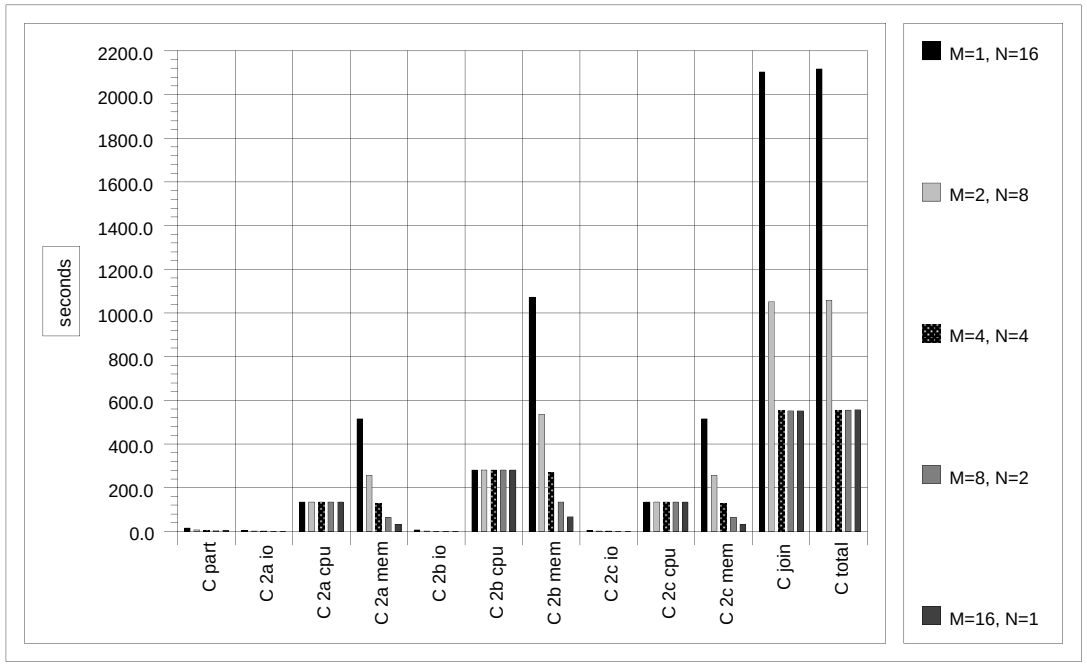


Figure 8.15: Dependency on architectural parameters $\mathcal{M}$ and $\mathcal{N}$ (Experiment 1).

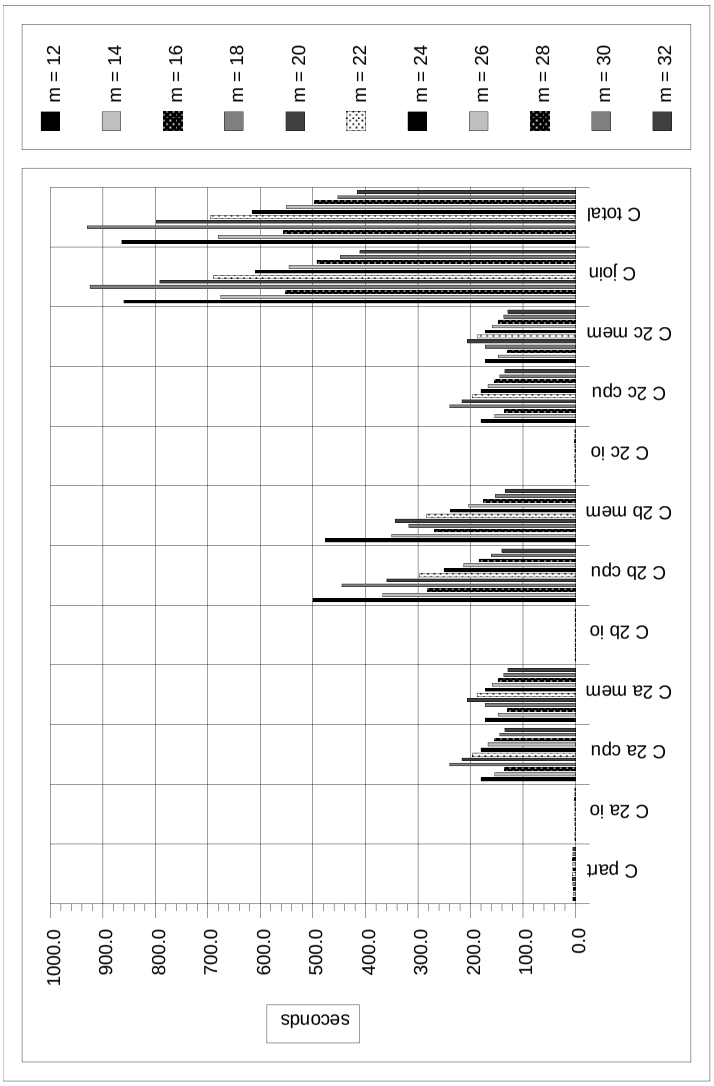


Figure 8.16: Dependency on the number m of partial joins (Experiment 2).

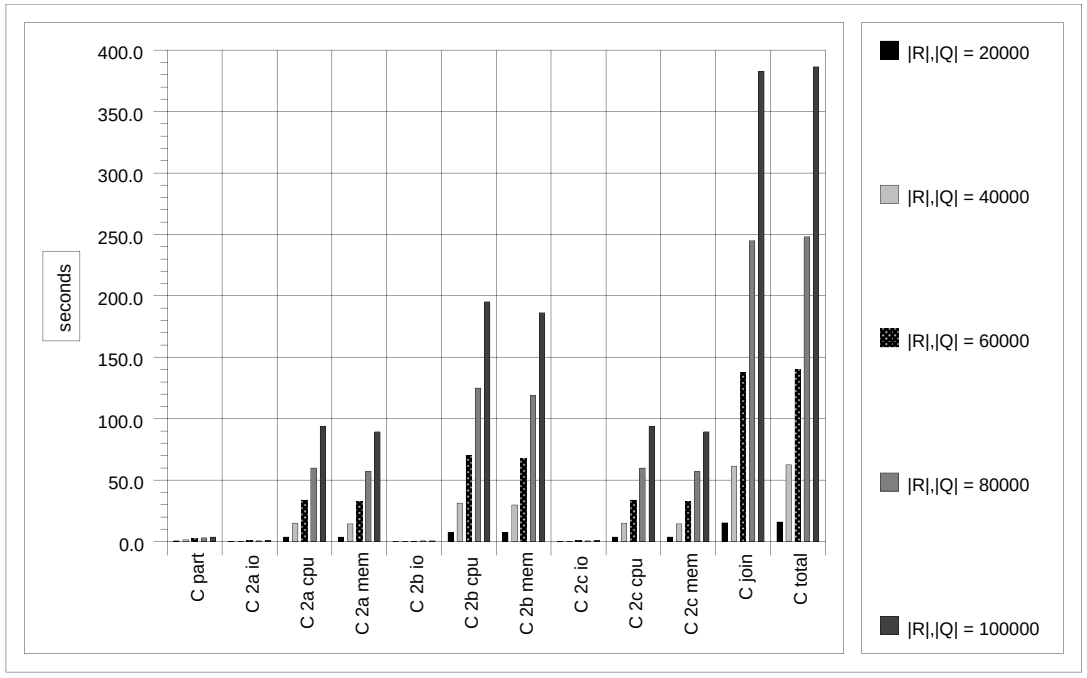


Figure 8.17: Dependency on the relations' sizes $|R|$ and $|Q|$ (Experiment 3).

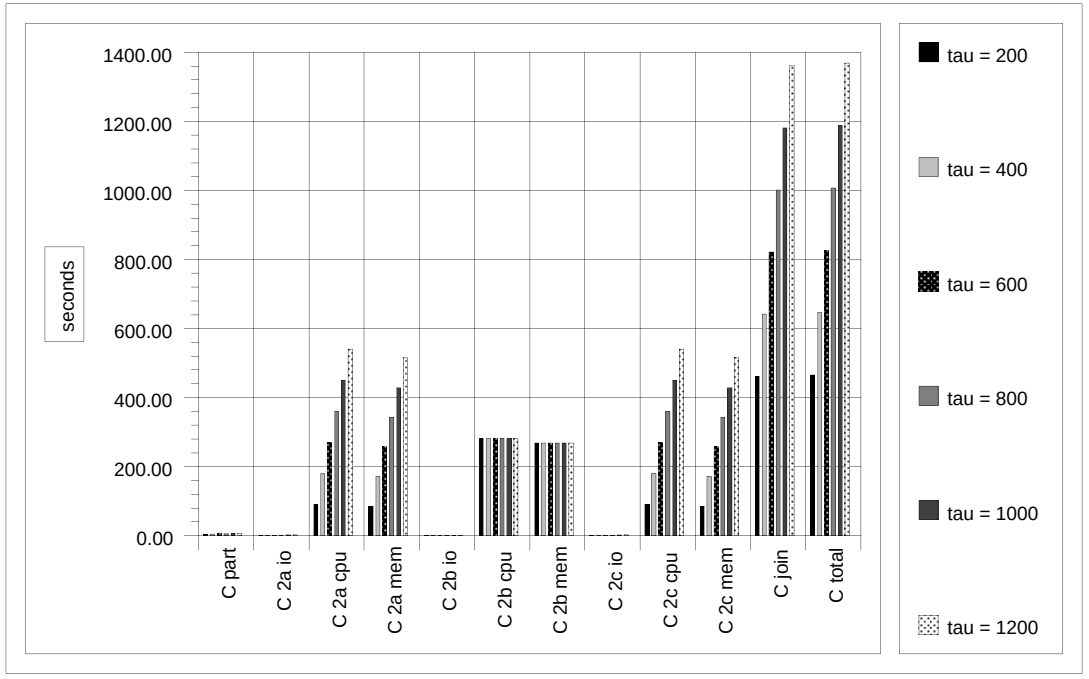
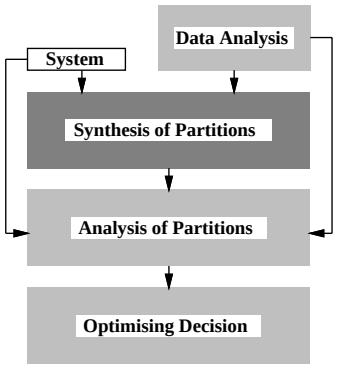


Figure 8.18: Dependency on the average interval length τ (Experiment 4).

Chapter 9

Partitioning Strategies

In this chapter, we present several families of partitioning strategies[1], all of which can be based on the information stored in an IP-table. These strategies create partitions $P = \{p_1, \ldots, p_{m-1}\}$ for computing a symmetrically partitioned temporal join (3.6):



$$R \bowtie_C Q \;=\; R_1 \bowtie_C Q_1 \cup \cdots \cup R_m \bowtie_C Q_m$$

of two relations R and Q. There are many goals according to which the fragments $R_1, \ldots, R_m, Q_1, \ldots, Q_m$ can be created. For example, one could aim to minimise the processing costs. However, this task is not that easy due to the complexity of the performance model (see chapter 8). Even relatively simple constraints, such as the ones for IP (see chapter 5), can necessitate a very expensive calculation in order to find a suitable partition. This leads us to consider alternative goals, such as the efficiency of the partitioning strategy itself.

In the following, several goals, and the family of strategies that result from it will be discussed. We thereby concentrate on the most general goals and strategies. All the algorithms that are used in that context van be efficiently implemented using IP-tables. In the remainder, we adopt the notation of complete IP-tables. Nevertheless, all of the techniques and algorithms that are described can be used in conjunction with incomplete IP-tables too, possibly at the expense of a decreased quality of the result. If this is the case we will point to this fact.

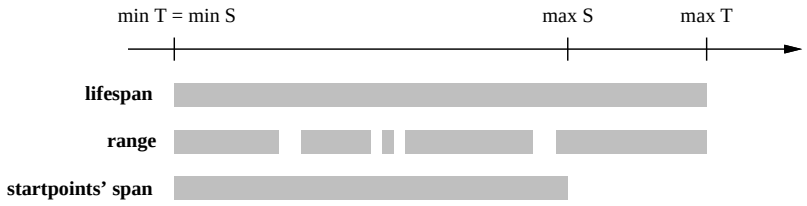
[1] In the remainder, we will frequently use the term *strategy* as a shortcut for *partitioning strategy*.

9.1 Uniform Strategies

This family comprises a number of a very simple strategies. Their common characteristic is that they divide a certain set of chronons – to which we refer as the *span* – into m disjoint segments, each of which containing the same number of chronons. The differentiating element is therefore the set of chronons that is used as the span; we discuss the following three and show how the corresponding partitions are computed by using IP-tables:

- the joint lifespan $L(R \cup Q)$ (see section 9.1.1),
- the joint range $T(R \cup Q)$ (see section 9.1.2),
- the startpoints' span $SP(R \cup Q) = [\min S(R \cup Q), \max S(R \cup Q)]$ (see section 9.1.3).

A comparison between the three types of spans is shown in figure 9.1 for some scenario of intervals; this comparison should make the differences obvious.



Notation: $T = T(R \cup Q), \quad S = S(R \cup Q)$

Figure 9.1: Comparison of the notions of a lifespan, a range and a startpoint span.

9.1.1 Uniform Lifespan Partitioning

The lifespan $L(R \cup Q)$ is simply the span between the first and the last timepoint in the IP-table $I(R \cup Q)$, i.e.

$$L(R \cup Q) = [t_1, t_N]$$

The length $|L(R \cup Q)|$ of the lifespan, i.e. the number of chronons, can therefore be calculated as

$$|L(R \cup Q)| = t_N - t_1$$

Partitioning the lifespan uniformly into m segments means that each segment should comprise

$$\frac{|L(R \cup Q)|}{m} \tag{9.1}$$

chronons and the breakpoints of the partition P are at that distance from each other. A breakpoint p_k is determined by

$$p_k \;=\; t_1 \;+\; \left\lceil k \cdot \frac{|L(R \cup Q)|}{m} \right\rceil$$

for $k = 1, \ldots, m-1$. This takes into account the possibility that the ratio in (9.1) might not result in an integer. Figure 9.2 summarises the algorithm for calculating a uniform lifespan partition. An example of such a partition is shown in figure 9.3 for the example scenario that has been used chapter 5.

```
/* V(R ∪ Q) = {t_1, ..., t_N} with t_j < t_{j+1} */

target = (t_N − t_1) / m          /* target length of the segments */

for k = 1 to m − 1 do
    p_k = t_1 + ⌈k · target⌉
od
```

Figure 9.2: Algorithm for partitioning $L(R \cup Q)$ uniformly.

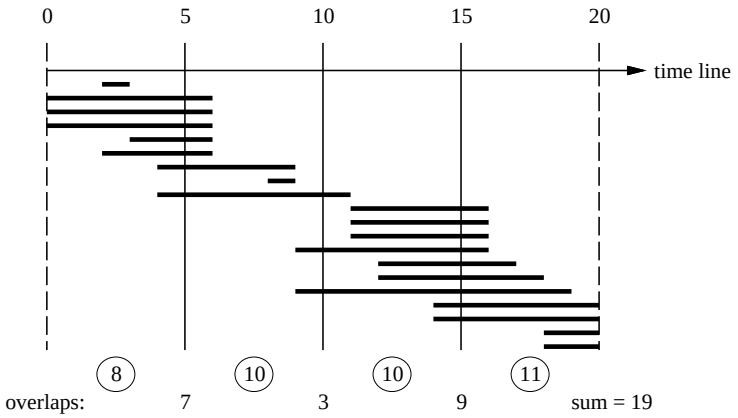


Figure 9.3: A uniform lifespan partition for the example of figure 5.2.

9.1.2 Uniform Range Partitioning

The main difference between the lifespan $L(R \cup Q)$ and the range $T(R \cup Q)$ is that the range does not contain those parts of the lifespan that are not covered by any interval in R or Q. In terms of the examples of chapter 7, where we analysed login-information to computers, this means that there might be times during which nobody was logged in to the computer(s), e.g. because of a down-time or a holiday. The idea behind partitioning the range rather than the lifespan is to omit such 'gaps' during which no temporal data is valid. Soo *et al.*, for example, use this approach[2].

The complete IP-table $I(R \cup Q)$ provides sufficient information to identify the gaps, i.e. those parts of $L(R \cup Q)$ that do not belong to $T(R \cup Q)$: the gaps are those areas between a t_{j-1} and a t_j that are not overlapped by any intervals, i.e. $o_{R \cup Q}(t_j) = 0$. Thus all entries in $I(R \cup Q)$ with a 0 entry in the third column identify a gap.

In order to determine the breakpoint of a uniform range partition one has to calculate the length $|T(R \cup Q)|$ of the range, compute the target length of each segment by

$$\left\lceil \frac{|T(R \cup Q)|}{m} \right\rceil$$

and finally determine the breakpoints p_k. Figure 9.4 summarises the algorithm. The uniform range partition for the example of figure 5.2 is identical with the uniform lifespan partition as shown in figure 9.3 because the lifespan and range are identical in that case.

Uniform range partitioning, as outlined in figure 9.4, works well with complete IP-tables. However, we might not be able to identify the gaps when using an incomplete IP-table because the condensation process might have collapsed a gap into a condensed timepoint (see sections 7.3.3 and 7.3.4). Therefore the result will be close to the one achieved by uniform lifespan partitioning.

In practice, one has to doubt whether uniform range partitioning achieves much better results than uniform lifespan partitioning as one can expect the range to be close to the lifespan in many applications. Also, algorithms for uniform range partitioning, such as the one in figure 9.4 or *ChooseIntervals* in [Soo et al., 1994], are very inefficient in comparison to the one in figure 9.2. This fact presumably outweighs the small benefit that can be drawn from partitioning the range rather than the lifespan.

[2] See algorithm *ChooseIntervals* in [Soo et al., 1994].

```
/* V(R ∪ Q) = {t_1, ..., t_N} with t_j < t_{j+1} */

rangelength = 0

for j = 2 to N do                          /* calculate |T(R ∪ Q)| */
   if o_{R∪Q}(t_j) > 0 then                /* consider only if there was no gap */
      rangelength = rangelength + (t_j − t_{j−1})
   fi
od

target = ⌈rangelength / m⌉    /* target length of the segments */
k      = 1                    /* number of next breakpoint p_k to be computed */
length = 0                    /* length of current segment */

for j = 2 to N do             /* scans the IP-table for R ∪ Q */
   if length > target then    /* length of current segment exceeds target length */
      p_k    = t_{j−1}
      k      = k + 1
      length = 0
   fi
   if o_{R∪Q}(t_j) > 0 then   /* consider length only if there was no gap */
      length = length + (t_j − t_{j−1})
   fi
od
```

Figure 9.4: Algorithm for partitioning $T(R \cup Q)$ uniformly.

9.1.3 Uniform Startpoints' Span Partitioning

As a third option, we propose to divide the startpoints' span

$$SP(R \cup Q) \;=\; [\min S(R \cup Q), \max S(R \cup Q)]$$

for the following reason: after $\max S(R \cup Q)$, no more intervals start, i.e. no more intervals are added to the plot. If a breakpoint p_k was chosen after that point then the fragments R_{k+1} and Q_{k+1} would hold only intervals that are already in R_k and Q_k and thus would already be joined in $R_k \bowtie Q_k$. Thus a join $R_{k+1} \bowtie Q_{k+1}$ would be without relevance. It is therefore feasible to divide the startpoints' span rather than the lifespan in order to avoid such a situation.

The only significant difference in comparison to uniform lifespan partitioning is that $|SP(R \cup Q)| \;\leq\; |L(R \cup Q)|$ and therefore that the lengths of the segments might be smaller. $|SP(R \cup Q)|$ can be calculated by using the IP-table $I(R \cup Q)$:

$$|SP(R \cup Q)| \;=\; t_S - t_1$$

with

$$t_S \;=\; \max\{t \in V(R \cup Q) : s_{R \cup Q}(t) > 0\} \;=\; \max S(R \cup Q)$$

The algorithm for determining the breakpoints is given in figure 9.5. Figure 9.6 shows the uniform startpoints' partition for the example of figure 5.2.

As in the case of uniform range partitioning, incomplete IP-tables might not provide sufficient information to determine the startpoints' span exactly. Nevertheless, the algorithm in figure 9.5 might still calculate a $t_S < t_N$ and therefore might still provide some of the benefits of the startpoints' span over the lifespan approach.

The index of a t_S within an IP-table can be stored as an additional parameter in order to avoid to compute it at run time: when two or more IP-tables are merged then the resulting t_S can be computed as

$$t_S \;=\; \max\{t_{S1}, t_{S2}, \ldots, t_{Sn}\}$$

where $t_{S1}, t_{S2}, \ldots, t_{Sn}$ are the respective maximum startpoints of the IP-tables that participate in the merge.

9.1.4 Conclusions

We conclude our discussion of uniform partitioning strategies considering two practical aspects:

```
/* V(R ∪ Q) = {t_1, ..., t_N} with t_j < t_{j+1} */

t_S      = max{t ∈ V(R ∪ Q) : s_{R∪Q}(t) > 0}
target   = (t_S − t_1) / m

for k = 1 to m − 1 do
    p_k = t_1 + ⌈k · target⌉
od
```

Figure 9.5: Algorithm for partitioning $SP(R \cup Q)$ uniformly.

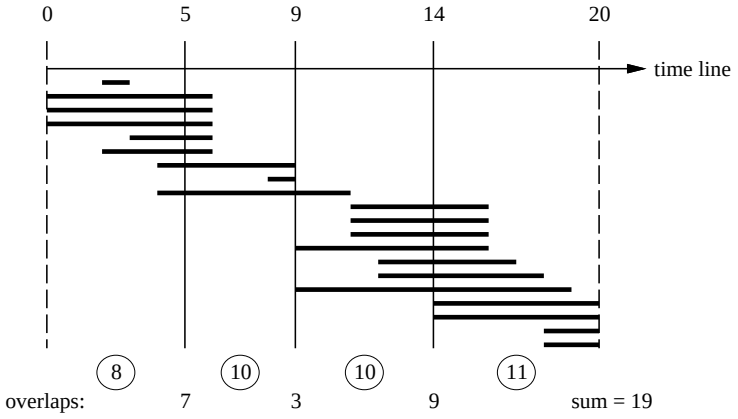


Figure 9.6: A uniform startpoints' span partition for the example of figure 5.2.

- It is doubtful if there is much difference between the lifespan, the range and the startpoints' span of temporal relations that are found in practice. Especially the significantly higher effort to determine the range from an IP-table in the algorithm of figure 9.4 will probably not pay off because many temporal relations have ranges that have only a few or no gaps. But it is the latter from which the benefits arise so the uniform range partitioning strategy is unlikely to provide a performance advantage that justifies the increased partitioning effort.

  In contrast to that, there is the uniform startpoints' span strategy that involves no computational disadvantage but offers a benefit in comparison to the uniform lifespan partitioning approach. The extent of this benefit is likely to be small if the lifespan is long and / or intervals are short. Both facts suggest that the startpoints' span is almost identical with the lifespan. However, the benefit might be marginal in other cases.

- It is obvious that the uniform strategies are liable to perform badly in the case of skewed, i.e. non-uniformly distributed, data. There is hardly any control over the fragments' loads and thus over the load balance. There are, however, many examples for temporal relations that have periodically repeated patterns of tuple timestamps: imagine a relation logging the starting and ending times of calls made by customers of a telephone company. The distribution of phone calls over a daytime period will vary significantly with many calls during business hours and few in the early morning and late evening. Considering a long time period, however, a daily pattern is probably repeated periodically. Thus we can expect a poor partition result when uniformly partitioning a one-day-span but a much better one for a period comprising several days, in particular when m matches the number of days. We will look into this issue further when we evaluate the partitioning strategies in chapter 10.

9.2 Underflow Strategies

The major disadvantage of the uniform partitioning strategies was the lack of control over the load balance. This deficit is overcome by the family of underflow strategies. Section 9.2.1 describes the algorithm for the basic strategy. In section 9.2.2, we discuss variations of that algorithm.

9.2.1 Basic Strategy

The idea of this strategy is to sequentially fill the fragments R_1, R_2, ... (Q_1, Q_2, ..., respectively) such that a given number X of tuples per fragment is 'underflowed', i.e. not exceeded. The IP-tables for the individual relations R and Q together with equation (7.1) can be used for this purpose.

The algorithm starts by filling up fragments R_1 and Q_1 by moving the first breakpoint, p_1, as far as possible, i.e. as long as $|R_1| \leq X$ and $|Q_1| \leq X$ is guaranteed. When p_1 is set, then R_2 and Q_2 are filled by moving the second breakpoint, p_2, as far as possible. The same process is repeated for the following fragments. The number m of fragments is thereby a result of the partitioning process rather than an input parameter as in the case of the uniform strategies.

Figure 9.7 summarises the algorithm that implements this strategy using the IP-tables of relations R, Q and $R \cup Q$. Figure 9.8 shows the partition resulting from partitioning the example of figure 5.2 using the basic underflow strategy with $X = 10$.

```
/* V(R ∪ Q) = {t_1, ..., t_N} with t_j < t_{j+1} */

k       = 1     /* number of next breakpoint p_k to be computed */
load_R  = 0     /* current load of fragment R_k */
load_Q  = 0     /* current load of fragment Q_k */

for j = 1 to N do
  if load_R + s_R(t_j) > X or load_Q + s_Q(t_j) > X then   /* |R_k| or |Q_k| would */
    p_k     = t_{j-1}                                      /* exceed the maximum */
    k       = k + 1                                        /* load X */
    load_R  = o_R(t_{j-1})
    load_Q  = o_Q(t_{j-1})
  fi
  load_R = load_R + s_R(t_j)
  load_Q = load_Q + s_Q(t_j)
od
```

Figure 9.7: Algorithm for the basic underflow strategy using the IP-tables relations $I(R)$, $I(Q)$ and $I(R \cup Q)$.

9.2.2 Variations

The algorithm in figure 9.7 controls the $|R_k|$ and $|Q_k|$ by guaranteeing that they do not exceed the limit X for all $k = 1, \ldots, m$. However, when discussing the

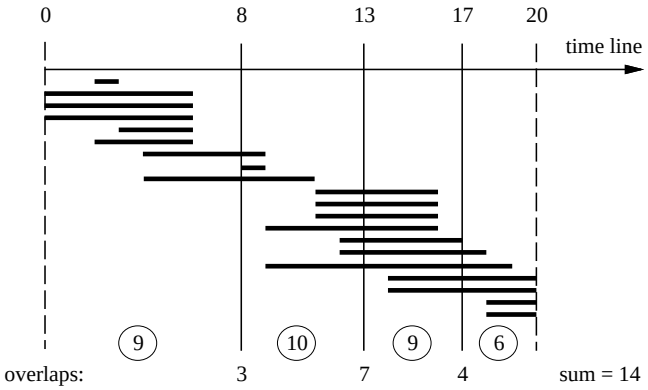


Figure 9.8: The partition for the example of figure 5.2 using the basic underflow strategy with a maximum load of $X = 10$.

performance model of chapter 8, we were mainly concerned with the numbers $|R'_k|$ and $|Q'_k|$ as we wanted them to fit into the local main memory of a processor if possible. But this can be easily achieved by keeping all $|R'_k|$ below a limit X_R and all $|Q'_k|$ below a limit X_Q whereby the limits can be determined by

$$\begin{array}{rcl} |R'_k| \cdot |r| & \leq & \frac{mem}{\mathcal{N}} \\ |Q'_k| \cdot |q| & \leq & \frac{mem}{\mathcal{N}} \end{array} \quad \Leftrightarrow \quad \begin{array}{rcl} |R'_k| & \leq & \frac{mem}{\mathcal{N} \cdot |r|} \\ |Q'_k| & \leq & \frac{mem}{\mathcal{N} \cdot |q|} \end{array} \quad = \quad \begin{array}{l} X_R \\ X_Q \end{array}$$

Alternatively, one can try to achieve a specific number m_{target} of fragments. Each tuple is assigned to exactly one primary fragment. Therefore one can expect a partition to create m fragments if

$$X_R \approx \frac{|R|}{m_{target}} \quad \text{and} \quad X_Q \approx \frac{|Q|}{m_{target}}$$

In practice, X_R and X_Q need to be slightly higher because most primary fragments cannot be filled up to X_R or X_Q, respectively. Consequently, the resulting m will probably be higher than m_{target}.

With X_R and X_Q being determined, the algorithm of figure 9.7 can be adapted to guarantee that $|R'_k| \leq X_R$ and $|Q'_k| \leq X_Q$ for all $k = 1, \ldots, m$. It is shown in figure 9.9.

Many more variations of that type can be designed. For example, one could try the limit products, such as $|R_k| \cdot |Q_k|$, $|R'_k| \cdot |Q'_k|$ etc., by a certain maximum X. These products are part of the most expensive cost components in the performance model of chapter 8. However, there is no clear indication of how to choose an adequate limiting value for X in that case. This makes such partitioning strategies difficult to handle. Presumably, X would be determined by performance experiments on an existing DBMS installation. Consequently, its

value would depend on rather installation-specific characteristics. We therefore do not expand on this issue any further.

The advantage of the underflow strategies is that the fragments' loads are well controlled and can be expected to be well balanced as long as the values $s_R(t_j)$ are relatively small in order to approach the limit X as close as possible (see if-conditions in figures 9.7 and 9.9). There is, however, no control over the number of intervals overlapping the breakpoints. In sections 9.3 and 9.4, we present two techniques that can be considered as enhancements of the underflow strategies that have been presented so far.

```
/* V(R ∪ Q) = {t_1, ..., t_N} with t_j < t_{j+1} */

k       = 1     /* number of next breakpoint p_k to be computed */
load_R  = 0     /* load of current fragment R'_k */
load_Q  = 0     /* load of current fragment Q'_k */

for j = 1 to N do
  if load_R + s_R(t_j) > X_R or load_Q + s_Q(t_j) > X_Q then   /* |R'_k| or |Q'_k| would */
    p_k     = t_{j-1}                                          /* exceed the resp. max. */
    k       = k + 1                                            /* loads X_R and X_Q */
    load_R  = 0
    load_Q  = 0
  fi
  load_R = load_R + s_R(t_j)
  load_Q = load_Q + s_Q(t_j)
od
```

Figure 9.9: Algorithm implementing the underflow strategy for the primary fragments R'_k and Q'_k.

9.3 Minimum-Overlaps Strategies

9.3.1 Basic Strategy

Similar to the underflow strategy, the goal of this one is to create fragments R_1, R_2, ... (Q_1, Q_2, ... , respectively) such that a given number X of tuples is not exceeded *and* that the sum of intervals overlapping the breakpoints, i.e.

$$\sum_{k=1}^{m-1} o_{R \cup Q}(p_k)$$

is minimal at the same time. But this looks very much like IP (see chapter 5). In fact, we can use a variation of `IP-opt` to compute such a partition. We note that the original version of `IP-opt` limits the total number of intervals (tuples) that fall into a segment, i.e. in terms of a join $R \bowtie_C Q$ it would use

$$|R_k| + |Q_k| \ \leq \ X$$

as the constraint to determine the breakpoints rather than

$$|R_k| \leq X \qquad \text{and} \qquad |Q_k| \leq X$$

for all $k = 1, \ldots, m$ as the basic underflow strategy.

The variation of `IP-opt` can be based on the information stored in the IP-table $I(R \cup Q)$. First, two arrays $\mathit{load}_R[j,i]$ and $\mathit{load}_Q[j,i]$ are initialised according to equation (5.10). Then the timepoints $t_1, \ldots, t_N$ of $V(R \cup Q)$ are processed from 1 to N. For each t_i, a partition is computed for the span $[t_1, t_i]$ that has minimal sum of overlaps and has fragment loads less than X. More precisely, for each t_i a predecessor $\mathit{pred}\,(t_i)$ is determined which represents the preceding breakpoint that leads to the partition with a minimum number of overlapping tuples. If no such predecessor can be found then the original `IP-opt` stops with the message that there is no partition that satisfies the constraints. For practical purpose, we propose to weaken this and to use the timepoint t_{i-1} (that precedes t_i) as $\mathit{pred}\,(t_i)$ despite the fact that the fragment resulting from that has a size larger than X. This is in line with the basic underflow strategy which copes with such an extreme situation in the same way.

Finally, a minimising partition can be obtained from the sequence $\mathit{pred}\,(t_N)$, $\mathit{pred}\,(\mathit{pred}\,(t_N))$, ... (until a delimiting dummy point t_0 appears). For a detailed discussion of `IP-opt` refer to section 5.5. Figure 9.10 summarises the algorithm in a form that makes use of IP-tables.

Figure 9.11 shows how the example scenario is partitioned using the minimum-overlaps strategy for $X = 10$. In comparison to the underflow strategy it reduces the total number of overlaps from 13 to 9. Therefore, one can expect this strategy to perform at least as well as the basic underflow strategy for single-processor systems. In the case of a multiprocessor setting, the advantage of having a reduced total number of overlaps might not necessarily pay off: the minimum-overlaps algorithm chooses breakpoints that reduce overlaps possibly at the expense of achieving well balanced fragments. In contrast, the underflow strategy aims to create equally filled fragments. Finally, another disadvantage is the algorithms run time complexity of $O(N^2)$ compared

to $O(N)$ of the underflow strategy. This implies that the minimum-overlaps strategy should only be applied to small IP-tables or to very big joins, i.e. in a situation in which the time spent on the optimisation through the minimum-overlaps strategy does contribute only marginally to the overall join costs.

9.3.2 Variations

Similarly to the variations for the basic underflow strategy one can design variations for the basic minimum-overlaps strategy. For example, one can aim to limit the $|R'_k|$ by X_R and the $|Q'_k|$ by X_Q as done in section 9.2.2 but this time also minimising the total number of overlapping tuples. This variation of basic algorithm of figure 9.10 is shown in figure 9.12.

The major problem of the two versions of the minimum-overlaps strategy is the run time complexity of $O(N^2)$. The only possibility to ease this problem is to decrease N, e.g. by using condensed or endpoint IP-tables. A further possible reduction method is the black-out preprocessing technique which is described in the following section.

```
/* V(R ∪ Q) = {t_1, ..., t_N} with t_j < t_{j+1} */

/* Use a dummy point t_0 */
o_{R∪Q}(t_0) = 0
c(t_0)       = 0

/* Initialise arrays load_R[j,i] and load_Q[j,i] according to equation (5.10) */
for j = 0 to (N − 1) do
    load_R[j,j] = o_R(t_j)
    load_Q[j,j] = o_Q(t_j)
    for i = (j + 1) to N do
        load_R[j,i] = load_R[j,i−1] + s_R(t_i)
        load_Q[j,i] = load_Q[j,i−1] + s_Q(t_i)
    od
od

/* Partitioning */
for i = 1 to N do
    c(t_i)      = ∞
    pred(t_i)   = ∞
    for j = 0 to (i − 1) do
        if load_R[j,i] ≤ X and load_Q[j,i] ≤ X and o_{R∪Q}(t_j) + c(t_j) < c(t_i) then
          c(t_i)      = o_{R∪Q}(t_j) + c(t_j)
          pred(t_i)   = t_j
        fi
    od
    if pred(t_i) = ∞ then
      /* no pred(t_i) has been found */
      c(t_i)      = o_{R∪Q}(t_{i−1}) + c(t_{i−1})
      pred(t_i)   = t_{i−1}
    fi
od

/* Create partition P */
p = t_N
P = ∅
while p ≥ t_1 do
     p = pred(p)
     P = P ∪ {p}
od
P = P − {t_0}
m = |P| + 1
```

Figure 9.10: Basic algorithm of the minimum-overlaps strategy for relations R and Q.

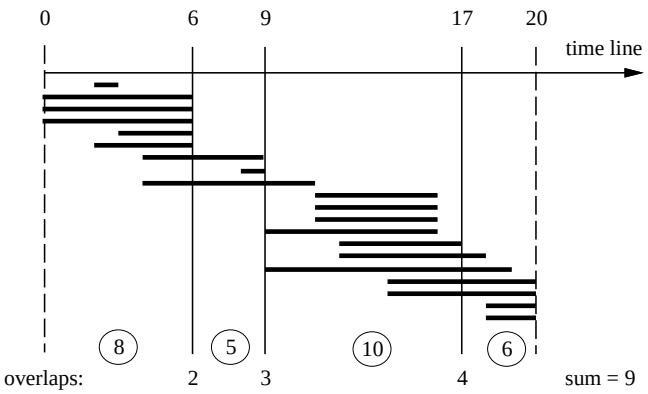


Figure 9.11: The partition for the example of figure 5.2 using the minimum-overlaps strategy with a maximum load of $X = 10$.

```
/* V(R ∪ Q) = {t_1, ..., t_N} with t_j < t_{j+1} */

/* Use a dummy point t_0 */
o_{R∪Q}(t_0) = 0
c(t_0)       = 0

/* Initialise arrays load_R[j,i] and load_Q[j,i] according to equation (5.10) */
for j = 0 to (N − 1) do
    load_R[j,j] = 0
    load_Q[j,j] = 0
    for i = (j + 1) to N do
        load_R[j,i] = load_R[j,i−1] + s_R(t_i)
        load_Q[j,i] = load_Q[j,i−1] + s_Q(t_i)
    od
od

/* Partitioning */
for i = 1 to N do
    c(t_i)      = ∞
    pred(t_i)   = ∞
    for j = 0 to (i − 1) do
        if load_R[j,i] ≤ X_R and load_Q[j,i] ≤ X_Q and o_{R∪Q}(t_j) + c(t_j) < c(t_i) then
          c(t_i)      = o_{R∪Q}(t_j) + c(t_j)
          pred(t_i)   = t_j
        fi
    od
    if pred(t_i) = ∞ then
      /* no pred(t_i) has been found */
      c(t_i)      = o_{R∪Q}(t_{i−1}) + c(t_{i−1})
      pred(t_i)   = t_{i−1}
    fi
od

/* Create partition P */
p = t_N
P = ∅
while p ≥ t_1 do
     p = pred(p)
     P = P ∪ {p}
od
P = P − {t_0}
m = |P| + 1
```

Figure 9.12: Algorithm of the minimal-overlaps strategy for limiting the primary fragments R'_k and Q'_k.

9.4 Black-Out Preprocessing Strategy

The underflow partitioning strategies do not pay any attention to the number of overlapping intervals when they choose a breakpoint. The incorporation of a mechanism that minimises the total number of overlaps led to the minimum-overlaps strategy. However, the latter suffers from an algorithmic complexity of $O(N^2)$ which is prohibitive for the high values of N that can be expected in practice.

In this section, we describe an alternative, but heuristical technique to reduce the number of overlaps. The idea is the following: figure 9.13 shows a typical example of a function[3] $o_R(t)$. The general goal is to find breakpoints p_k for which $o_R(p_k)$ is low, i.e. somewhere in the valleys formed by $o_R(t)$. Therefore one could restrict a strategy's choice to those timepoints, i.e. 'cut out' the unfavourable bits of the time domain.

In practical terms, we can do this in the following way: the IP-table $I(R)$ is scanned and all $t_j \in V(R)$ with $o_R(t_j) > Y$ are blacked out. Y is called the *black-out threshold*. This process creates a new IP-table $\bar{I}(R)$ and is very similar to the condensation process that was described in section 7.3.3. Figure 9.14 summarise the basic algorithm for this *black-out strategy*.

Figure 9.15 shows the result of the black-out strategy when it is applied to $o_R(t)$ for $R =$ *EPCC* (week-lifespan; see section 7.3.2) with

$$Y \;=\; \frac{1}{N} \cdot \sum_{j=1}^{N} o_R(t_j)$$

i.e. the average of o_R. The parts of $V(R)$ that are cut out are marked as black bars on the time axis. If the maximum load X (X_R, X_Q respectively) in an underflow strategy is high enough then breakpoints can be chosen to put all the tuples that are valid within one of the time periods marked by the bars to be put into one fragment. If this is not the case for only one of the 'bar-periods' then an underflow strategy cannot find an allowable partition because of the black-out preprocessing. Such a situation can arise if the black-out preprocessing creates long 'bar-periods', for example as shown in figure 9.16 for a different $o_R(t)$. There are two possibilities to shorten a long 'bar-period':

- The threshold Y can be increased.

[3] Here, we refer to $o_R(t)$ assuming that R is to be partitioned. However, the technique that we describe can be applied to any temporal relation, in particular also to $R \cup Q$ in which case $o_{R \cup Q}(t)$ would have to be used rather than $o_R(t)$.

- The 'bar-period' is split by admitting some t_j that fall into this period despite the fact that $o_R(t_j) > Y$.

The first possibility is not very attractive because it is difficult to determine by how much Y should be increased in order to guarantee an underflow strategy to be able to find a breakpoint when it is reaches the maximum load X. Apart from that the advantages of the black-out strategy are gradually lost when increasing Y.

The second possibility suggests that a long 'bar-period' is split into pieces that can be handled by an underflow strategy. Such pieces can be created by checking the load that is created by cutting out timepoints $t_j \in V(R)$ if $o_R(t_j) > Y$. If such a load would exceed a certain threshold Y' then a t_j is inserted into $\bar{I}(R)$ even if $o_R(t_j) > Y$. This advanced version of the black-out strategy is summarised in figure 9.17. An example of two additional timepoints being admitted within the 'bar-period' is shown in figure 9.18.

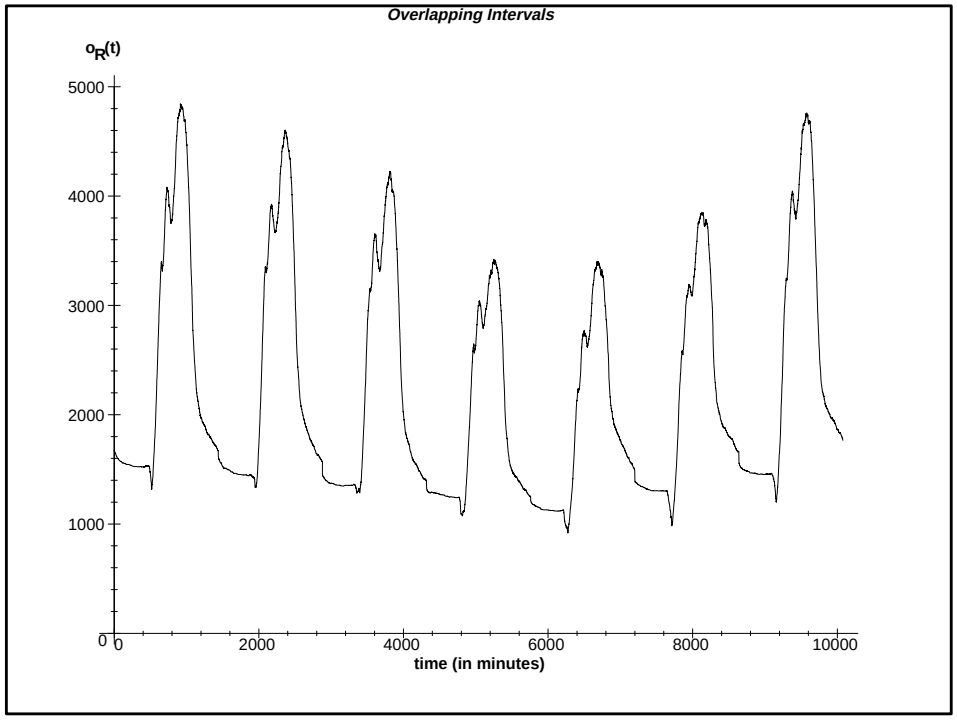


Figure 9.13: The function $o_R(t)$ for the temporal relation $R =$ *EPCC* (week-lifespan; see section 7.3.2).

```
/* Basic Black-Out Preprocessing for I(R) to create Ī(R) */

starts  = 0
Ī(R)    = ∅

for j = 1 to N do
    starts = starts + s_{R∪Q}(t_j)
    if j = 1 or j = N or o_{R∪Q}(t_j) ≤ Y then
        insert (t_j, starts, o_R(t_j)) into Ī(R)
        starts = 0
    fi
od
```

Figure 9.14: Basic black-out preprocessing for $I(R)$.

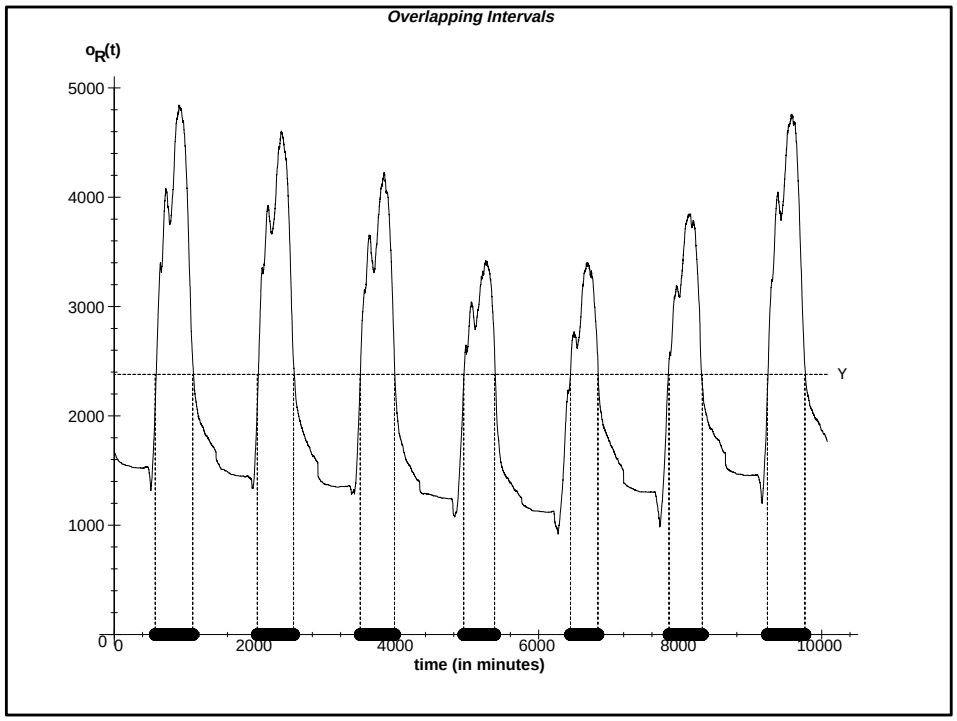


Figure 9.15: Black-out strategy applied to $o_R(t)$ of figure 9.13.

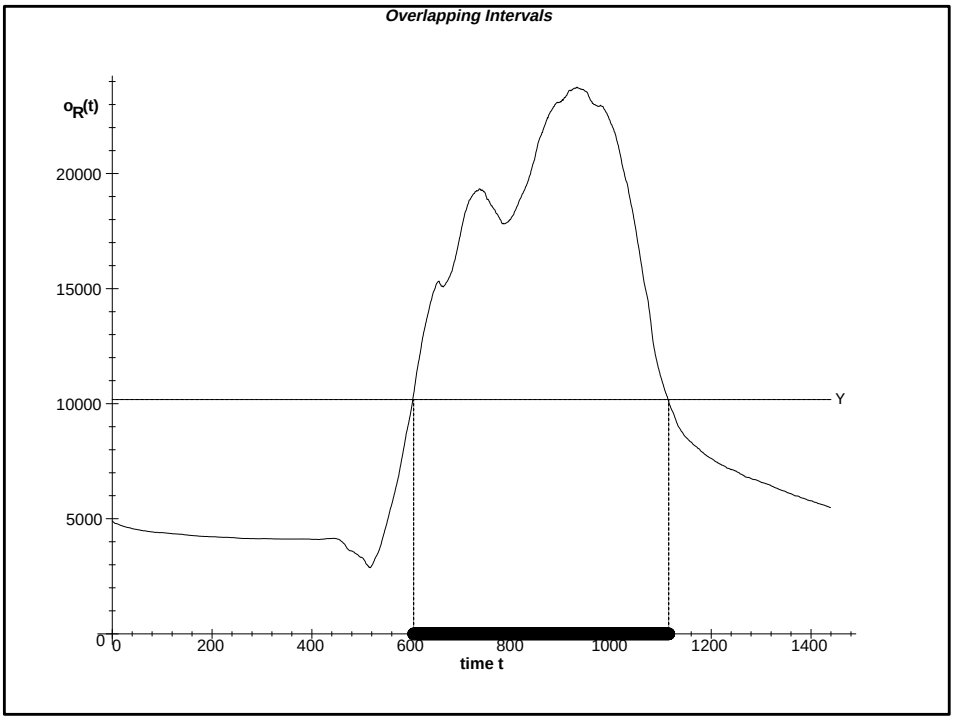


Figure 9.16: Black-out strategy applied to $o_R(t)$ for $R = \textit{EPCC}$ (day-lifespan; see section 7.3.2).

```
/* Advanced Black-Out Preprocessing for I(R) to create Ī(R) */

Ī(R)    = ∅
starts  = 0

for j = 1 to N do
    starts = starts + s_{R∪Q}(t_j)
    if j = 1 or j = N or o_{R∪Q}(t_j) ≤ Y then
        insert (t_j, starts, o_R(t_j)) into Ī(R)
        starts   = 0
        j_prev   = j
    else
    if (o_R(t_{j_prev}) + starts) > Y' then
        if j_prev ≠ (j − 1) then
            insert (t_{j−1}, (starts − s_R(t_j)), o_R(t_{j−1})) into Ī(R)
            starts   = o_R(t_{j−1}) + s_R(t_j)
            j_prev   = j − 1
        else
            output "Error. Cannot break up bar-period."
        fi
    fi
od
```

Figure 9.17: Advanced black-out preprocessing for $I(R)$.

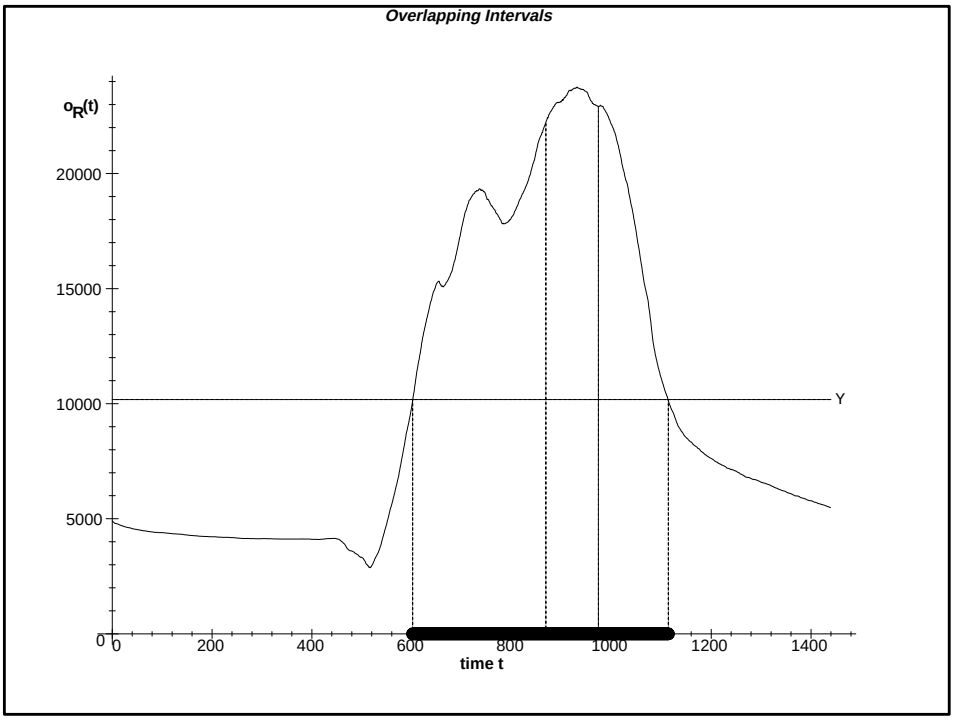


Figure 9.18: Advanced black-out strategy applied to $o_R(t)$ for $R =$ *EPCC-day* (see section 7.3.2).

Chapter 10

Experimental Evaluation

In this chapter, we evaluate the process for optimising partitioned temporal joins – as proposed in chapter 6 and elaborated in chapters 7 to 9. The experiments will focus on the various features that have been discussed so far and will follow the same path as the preliminary evaluation in section 8.5 which assumed uniform test data. Here, we will use real temporal data that was extracted from existing temporal relations; section 10.1 describes these data sets in more detail. In section 10.2 the performances of the various strategies of chapter 9 are compared in order to identify the most promising ones on which we can concentrate in the remaining experiments. This reduces the complexity of the following experiments significantly and makes the results easier to visualise and to interpret. In section 10.3, we look at the problem whether it is better to partition a join into many small or into a few but slightly bigger join computations. In other words, we vary the parameter m. While m is an input parameter for the uniform partitioning strategies it is an output parameter for the underflow and minimum-overlaps strategies. There, it is imposed by parameters X or X_R, X_Q which set maximum sizes for the various fragments or subfragments. In section 10.4 we therefore try to find a rule that allows us to determine the best-performing value for these parameters. In section 10.5, we look at the influence of the average interval length τ on the performances and whether certain values of τ favour certain partitioning strategies. In section 10.6, the sizes of the participating relations are varied. In section 10.7, we return to the question of the best-performing mixture $\mathcal{M}/\mathcal{N}$ of SMP-nodes and processors per SMP-node. We have already looked at this problem when conducting the experiments for uniform data. Here, we use skewed and somehow more realistic data. In sections 10.8 and 10.9, we look at the influences of condensation and black-out preprocessing on the performances. Finally, the main results are summarised in section 10.10.

10.1 The Test Data

10.1.1 Introduction

Timestamps of a temporal relation are influenced by various statistical processes. Let us re-consider the phone calls scenario: the start times are dictated by many factors such as

- daily routines, eg. the times when we wake up, work, have lunch, sleep etc.,
- business hours,
- the fact whether it is a working day or a public holiday, etc.

Furthermore, the lengths of the phone calls are a result of pricing or the nature of the calls, e.g. business calls as opposed to calls to a friend or a relative. Possibly, calls in the evenings are generally longer than daytime calls because of lower prices or because one tends to chat longer with friends or relatives rather than customers, bank managers, travel agents etc.

This is only one of many examples that illustrate how a set of 'real life' timestamps can be the result of a variety of statistical processes. We note that this feature is not restricted to transaction time but applies to many valid time scenarios as well. Just imagine the bookings database of a travel agent, travel organiser, car rental company or a hotel. Here, start and end times, i.e. the timestamp intervals, are dictated by dates for holiday seasons, public holidays or sports / theatre / music events, by special, promotional offers and possibly even by the weather.

The high statistical complexity behind the creation of timestamps is a significant difference in comparison to atomic data. It is therefore much more difficult to artificially create temporal test data with realistic properties. In the case of atomic data, many situations with a non-uniform distribution of the attribute values (i.e. data skew) have been successfully modelled using a Zipf distribution [Zipf, 1949]. An example of a paper that describes such experiments is [Wolf et al., 1993]. A similar approach for temporal data would either be

- unrealistic, if the statistical model is too simplistic, or
- too complex because a huge number of statistical parameters would have to be used; the underlying combinatorial effect would cause the experiments to be very hard to manage and to evaluate.

For these reasons, we decided to take an alternative approach for our experiments. It is based on real temporal data that we manipulate in order to control the experiments. The following section describes the data set and the manipulations that were performed.

10.1.2 The Basic Data Set

In section 7.3.2, we already used several sets of real temporal data in order to get an idea about realistic sizes of IP-tables. One of these sets was login information about accesses to a supercomputer system at the Edinburgh Parallel Computing Centre (EPCC). The set comprises accesses over a period of approximately five years. We used this set as a base for creating test data for the experiments in this chapter. The initial set was in a form as shown in figure 10.1 and contained over 125000 entries. Lines that did not contain any suitable information – such as those marked with a * in figure 10.1 – were deleted. This process left us with a set of 121728 entries. The latter were translated into two temporal relations to which we will refer as R and Q. In both relations, timestamps' start- and endpoints are integer values. The relations' lifespans are the same, ranging from 0 to 10079. The intention behind this range is that it corresponds to a week-long period in terms of minutes: $7 \cdot 24 \cdot 60 = 10080$. The differences between R and Q are the following:

- R has, what we call, a *periodic profile*. This means that the function $i_R(t)$ – which shows the number of tuples that have a timestamp that intersects with time t – consists of a pattern that is periodically repeated. In the case of the login data, one can assume that the login behaviour of the users repeats itself every day with weekends showing a reduced number of accesses. We can expect a similar profile in many other example scenarios such as the distribution and lengths of phone calls over a period of *several* days or holiday bookings (assuming yearly repeated patterns in the latter case). Figure 10.2 shows the periodic profile $i_R(t)$ of R. It will be discussed below in some more detail.

  The timestamp intervals of R were created from the login information in the following way: of each line, only the weekday and the start and end times were used. Times on Mondays were converted to numbers $0 \ldots 1439$[1], times on Tuesdays to $1440 \ldots 2879$, times on Wednesday to $2880 \ldots 4319$, etc. For example the line

[1]The number of minutes per day is $1440 = 24 \cdot 60$.

```
root   ttyp3   yanis.epcc.ed.a  Sun Oct 27 11:45 - 11:46  (00:00)
```

results in the interval

$$[\underbrace{6 \cdot 1440 + 11 \cdot 60 + 45}_{\text{Sun 11:45}}, \underbrace{6 \cdot 1440 + 11 \cdot 60 + 46}_{\text{Sun 11:46}}] = [9345, 9346]$$

The source code for the PERL script that converts login data as in figure 10.1 can be found in appendix B.1.

- Q has, what we call, a *non-periodic profile*. This means that the function $i_Q(t)$ – which shows the number of tuples that have a timestamp that intersects with time t – does not show patterns that are periodically repeated. In the case of the login data, one can assume that the login behaviour of the users during *one* day is non-periodic: from early morning onwards, there is a gradually growing number of users logging into the system. In the afternoon, this number starts to decrease with only a few users being logged in during the night. A similar scenario is again the distribution and lengths of phone calls during a *single* day. Figure 10.3 shows the non-periodic profile $i_Q(t)$ of Q. It will be discussed below in some more detail.

  The timestamp intervals of Q were created from the login information in the following way: of each line, only the start and end times were used. These times were converted to minutes of the day, i.e. mapped to a range $0 \ldots 1439$:

  $$\text{hh:mm} \rightarrow \text{hh} \cdot 60 + \text{mm}$$

  In a second step, these times were mapped to the range $0 \ldots 10079$ by multiplying them with 7 and adding a random number between 0 and 6. The random number avoids all interval start- and endpoints being multiples of 7. As an example, we consider again the line

  ```
  root   ttyp3   yanis.epcc.ed.a  Sun Oct 27 11:45 - 11:46  (00:00)
  ```

  which results in the interval

  $$[(11 \cdot 60 + 45) \cdot 7 + \text{rand}(0 \ldots 6), (11 \cdot 60 + 46) \cdot 7 + \text{rand}(0 \ldots 6)]$$
  $$\stackrel{\text{eg.}}{=} [4939, 4941]$$

  where *rand(0 ... 6)* is supposed to randomly choose a number from the set $\{0, 1, 2, 3, 4, 5, 6\}$. We note that the result is not deterministic because of the random numbers. The source code for the PERL script that converted login data as in figure 10.1 can be found in appendix B.2.

```
yuh    ftp     alab-16.ed.ac.u  Sun Oct 27 12:03 - 12:03  (00:00)
root   ttyp3   yanis.epcc.ed.a  Sun Oct 27 11:45 - 11:46  (00:00)
yuh    ftp     house.ed.ac.uk   Sun Oct 27 11:36 - 11:36  (00:00)
yuh    ttyp2   alab-16.ed.ac.u  Sun Oct 27 11:32 - 17:05  (05:33)
zxa    ttyp0   bottle.ph.ed.ac  Sun Oct 27 10:42 - 16:07  (05:24)
reboot console                  Sun Oct 27 09:30                    *
yuh    ttyp3   alab-16.ed.ac.u  Sun Oct 27 09:21 - down   (00:08)  *
onb01  ttyp0   aborg.dcs.st-an  Sun Oct 27 08:03 - 08:05  (00:01)
onb01  ttyp0   aborg.dcs.st-an  Sun Oct 27 07:52 - 07:56  (00:03)
yuh    ftp     house.ed.ac.uk   Sat Oct 26 18:47 - 18:48  (00:00)
smith  ttyp1   lilly.glg.ed.ac  Sat Oct 26 18:46 - 08:46  (13:59)
smith  ttyp1   lilly.glg.ed.ac  Sat Oct 26 17:20 - 18:45  (01:24)
```

Figure 10.1: An extract of the original login information.

The procedures that created the two collections of interval timestamps, R and Q, achieved different profiles. However, the procedure for Q mapped the original range of $0 \ldots 1439$ to one of $0 \ldots 10079$. This leads also to an increase in the lengths of the intervals. In fact, the average length of an interval τ_R in R is 118.5 (minutes) so far, whereas in Q we find $\tau_Q = 512.9$ (minutes). With respect to the join performance, this difference could subsume any effect that is caused by the different profiles. In order to avoid this, we applied an additional procedure *change_lengths()* to bring τ_R and τ_Q in line, namely to a value of 300 (minutes). The source code for *change_lengths()* is given in appendix C. Essentially, it randomly picks intervals and adds or deletes chronons from them until the desired average length is achieved. We will use this procedure also for controlling the experiments in section 10.5.

The final profiles for R and Q are respectively shown in figures 10.2 and 10.3. $i_R(t)$ has seven peeks corresponding to the daytime hours of the seven weekdays Monday, Tuesday, ..., Sunday. As one can expect, there are less accesses during Saturdays and Sundays: the two rightmost peeks are significantly lower than the previous ones. In contrast, $i_Q(t)$ describes the accesses during a day (if we ignore the values of the time axis for a moment): as one can expect, there is a sharp rise during the morning, with a little valley during lunch time. In the afternoon there is a second peek, followed by a sharp fall towards the evening. As we see from this interpretation of the profiles, there is a large number of factors that contribute to their shapes. This underlines the presence of a high statistical complexity that we can expect in many scenarios.

Table 10.1 summarises the main characteristics of R and Q. We will use R and Q as the base for the experiments; some of the parameters, however, will be varied such as τ_R and τ_Q (section 10.5) or $|R|$ and $|Q|$ (section 10.6). We note that the parameters shown in table 10.1 approximately match those that were used for the uniform data experiments in section 8.5.2. Similarly, we assume

the architectural parameters listed in table 10.2 which correspond to those in table 8.9. A parallel architecture with $\mathcal{M} = 4$ and $\mathcal{N} = 4$ and a single processor architecture ($\mathcal{M} = 1$ and $\mathcal{N} = 1$) will be used in the experiments.

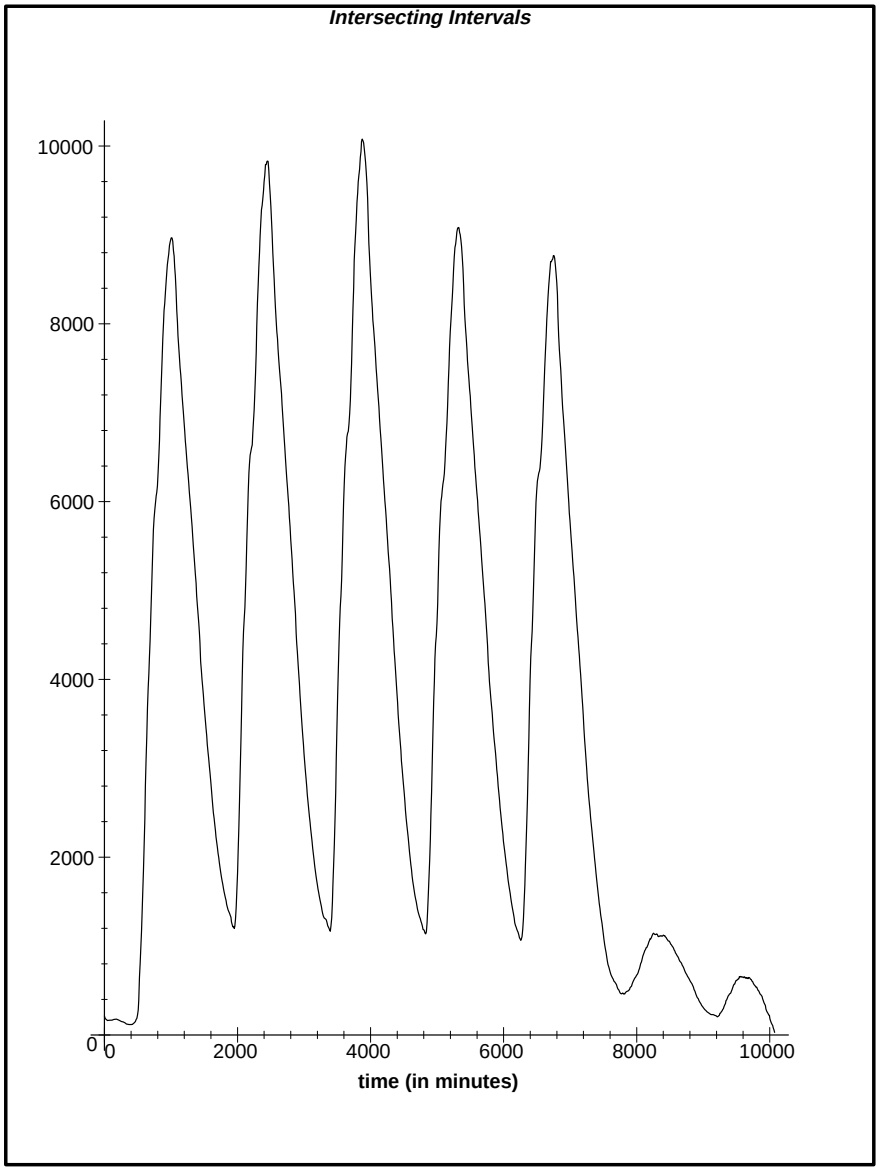


Figure 10.2: The periodic profile $i_R(t)$ of R.

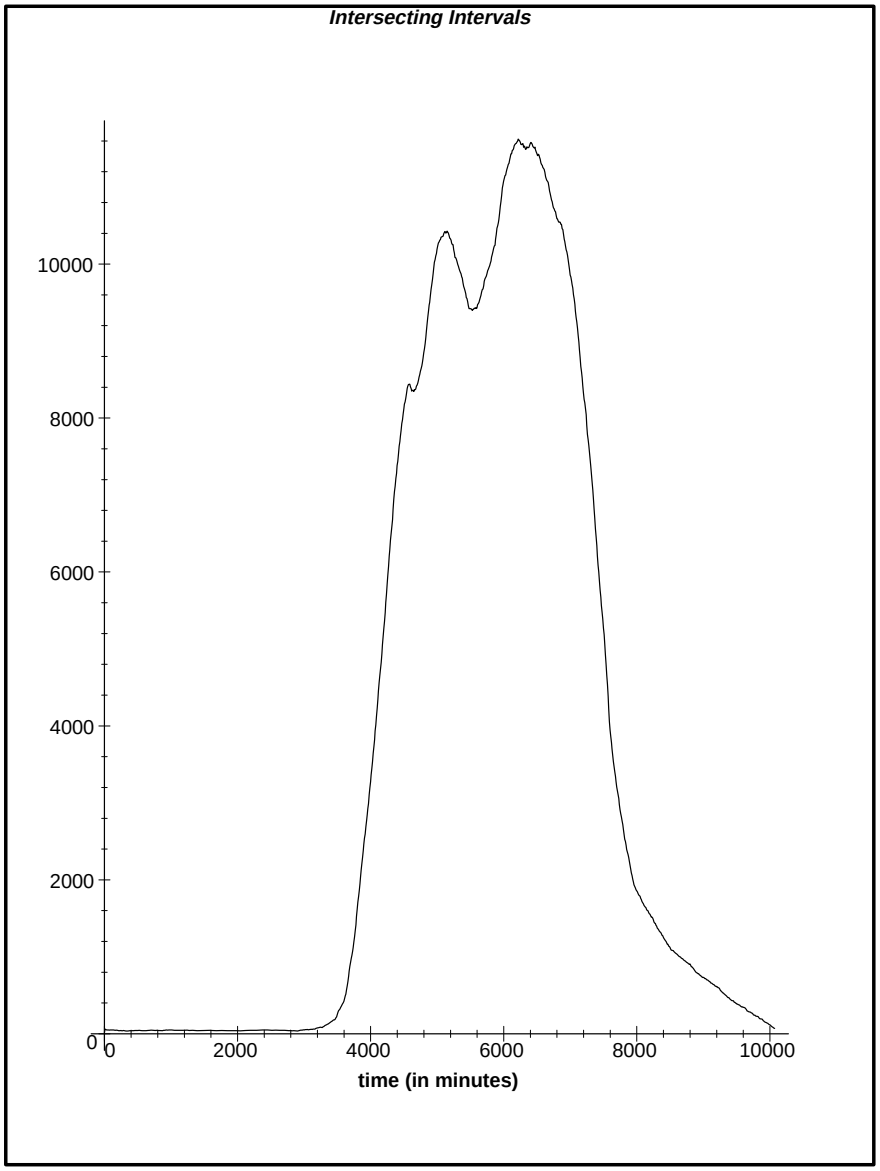


Figure 10.3: The non-periodic profile $i_Q(t)$ of Q.

Parameter	R	Q
size (in tuples)	$\|R\| = 121728$	$\|Q\| = 121728$
profile	periodic	non-periodic
lifespan (in minutes)	$\|L(R)\| = 10080$	$\|L(Q)\| = 10080$
τ (in minutes)	$\tau_R = 300$	$\tau_Q = 300$
tuple size (in bytes)	$\|r\| = 500$	$\|q\| = 500$

Table 10.1: The characteristics of the base relations R and Q.

Parameter	Description	Value
μ	processor speed in MIPS	200 MIPS
mem	free main memory per node	32 MB
w_{io}	disk I/O bandwidth per node	20 MB/sec
w_{com}	communication bandwidth	40 MB/sec
w_{mem}	memory bandwidth per node	400 MB/sec
I_{proc}	number of CPU instructions for processing a tuple in each step	1000
I_{hash}	number of CPU instructions for hashing a tuple	1000
I_{com}	number of CPU instructions for initiating a data transfer	500
I_{io}	number of CPU instructions for initiating a disk I/O	500
b	page size	4 kB

Table 10.2: The parameters describing the architecture that is used in the experiments.

10.2 A General Comparison between the Strategies

In chapter 9, several families of partitioning strategies have been discussed. In this section, we want to obtain some insight about their performance characteristics.

For that purpose, they were used to process the three joins $R \bowtie_C R$, $R \bowtie_C Q$ and $Q \bowtie_C Q$[2] where the join condition C simply requires the timestamp intervals to intersect. We note that the three joins have the profiles $i_R(t) \cdot i_R(t)$, $i_R(t) \cdot i_Q(t)$ and $i_Q(t) \cdot i_Q(t)$, i.e.

- the join $R \bowtie_C R$ has a periodic profile (figure 10.4),
- the join $R \bowtie_C Q$ has a partially periodic profile (figure 10.5),
- the join $Q \bowtie_C Q$ has a non-periodic profile (figure 10.6).

In total, we tested the following 11 strategies; each of them has been discussed in chapter 9:

1. *Uniform Lifespan*: this strategy uniformly partitions the joint lifespan of the participating relations (see section 9.1.1).
2. *Uniform Range*: this strategy uniformly partitions the joint range of the participating relations (see section 9.1.2).
3. *Uniform Startpoints Span*: this strategy uniformly partitions the joint startpoints span of the participating relations (see section 9.1.3).
4. *Basic Underflow*: this is the basic underflow strategy as described in section 9.2.1.
5. *Basic Underflow with b/o*: this is the basic underflow strategy (section 9.2.1) used in conjunction with the black-out preprocessing strategy (section 9.4).
6. *Primary Underflow*: this is the variation that applies underflow partitioning to the primary subfragments only (section 9.2.2).
7. *Primary Underflow with b/o*: this is the variation that applies underflow partitioning to the primary subfragments only (section 9.2.2). used in conjunction with the black-out preprocessing strategy (section 9.4).
8. *Basic Minimum-Overlaps*: this is the basic minimum-overlaps strategy as described in section 9.3.1.

[2]Sometimes we will refer to these joins also as *join 1*, *join 2* and *join 3* respectively.

9. *Basic Minimum-Overlaps with b/o*: this is the basic minimum-overlaps strategy (section 9.3.1) used in conjunction with the black-out preprocessing strategy (section 9.4).

10. *Primary Minimum-Overlaps*: this is the variation that applies minimum-overlaps partitioning to the primary subfragments only (section 9.3.2).

11. *Primary Minimum-Overlaps with b/o*: this is the variation that applies minimum-overlaps partitioning to the primary subfragments only (section 9.3.2). It is used in conjunction with the black-out preprocessing strategy (section 9.4).

We note that this list is not complete: in practice there can be many more partitioning strategies, e.g. further variations within the three families or some that take system-specific performance characteristics into account.

Table 10.3 shows the performance results for the strategies when being applied to the three joins and using a parallel and a single-processor hardware architecture. In order to guarantee a fair comparison between the strategies, the parameters X, X_R and X_Q for the underflow and minimum-overlaps strategies were chosen to produce $m = 16$ fragments / partial joins – the same number as the uniform strategies. We note that in the case of the parallel architecture ($\mathcal{M} = 4$, $\mathcal{N} = 4$) this means that each processor processes one partial join. For the black-out preprocessing strategy we used the average $\bar{O}$ of the $o_R(t)$ values in an IP-table $I(R)$ to be the respective threshold value Y, i.e.

$$Y \;=\; \bar{O} \;=\; \frac{1}{N} \cdot \sum_{j=1}^{N} o_R(t_j) \tag{10.1}$$

We will now look at certain issues that are 'hidden' within the many numbers in table 10.3. First, we want to get a general idea about the strategies, irrespective of the type of join. For that purpose, each of the numbers in table 10.3 is normalised in the following way: the performance results of the *uniform lifespan strategy* are respectively used to represent a general value 100. Columnwise, the times are converted into ratios with respect to the time achieved with the uniform lifespan strategy:

$$\frac{\text{time of strategy } X \text{ for join } n}{\text{time of uniform lifespan strategy for join } n} \cdot 100$$

For each of the two architectures, we then take the average of the three performance results per strategy. This normalisation guarantees that each join

contributes an equal share to the average and it is not the most expensive join that contributes most. Figure 10.7 shows the averages for the parallel and figure 10.8 for the single-processor architecture.

In the case of the parallel architecture, the primary underflow strategies are the clear winners, causing only around a third of the costs compared with the uniform strategies. The other underflow and the minimum-overlaps strategies end up in the area between 50 and 60. Apart from the quantitative aspect, this result is not surprising as the principal goal of the underflow strategies is to achieve a good balance between the fragments, regardless of the number of overlapping intervals. This means that all partial joins are more or less of equal sizes – a fact that is beneficial in the context of parallelism. The minimum-overlaps strategies make concessions with respect to the load balance in order to achieve a minimum-number of overlaps. These concessions obviously do not pay off. An interesting difference between the four underflow and the four minimum-overlaps strategies is that the primary underflow variations perform better than the basic strategies whereas the opposite relationship can be found between the minimum-overlaps strategies. In theory, one would expect the primary strategies to perform better in both cases because they take various aspects of the cost model into consideration (see discussions in sections 9.2.2 and 9.3.2). To explain the contrary effect in the case of the minimum-overlaps strategies, we have to look at the absolute figures in table 10.3: the primary versions of the minimum-overlaps strategies are also better for the joins $R \bowtie_C Q$ and $Q \bowtie_C Q$ but are somewhat far behind in the join $R \bowtie_C R$. This 'destroys' an otherwise favourable average value as used in figure 10.7.

There are two more conclusions that can be drawn from the diagram in figure 10.7:

- it confirms what we already expected in section 9.1, namely that there is not much difference between the uniform strategies;
- there does not seem to be much benefit that can be drawn from black-out preprocessing if there is a benefit at all. We will return to this problem in section 10.9 where experiments are conducted to analyse the black-out preprocessing strategy in more detail.

Now, we turn our attention to the figures obtained from the single-processor architecture (figure 10.8). Here, the scene looks quite different: the underflow and minimum-overlaps strategies perform at around 82-86% of the costs of

the uniform strategies. The basic minimum-overlaps strategy using black-out preprocessing is a narrow winner. Through the experiments in section 10.3, we will see that this scenario changes for higher values of m.

Finally, the question arises whether the optimisation process itself is efficient. It would not be worth while to optimise the partitioning if the optimisation itself imposed considerable costs. Figure 10.9 shows the average elapsed times that were spent on the optimisation itself. The hardware platform was a Sun SPARC SS-20 computing server with two processors. The uniform and underflow strategies required less than 2 seconds, whereas the minimum-overlaps strategies took around 16 seconds which is still relatively low in comparison to the join processing times in table 10.3.

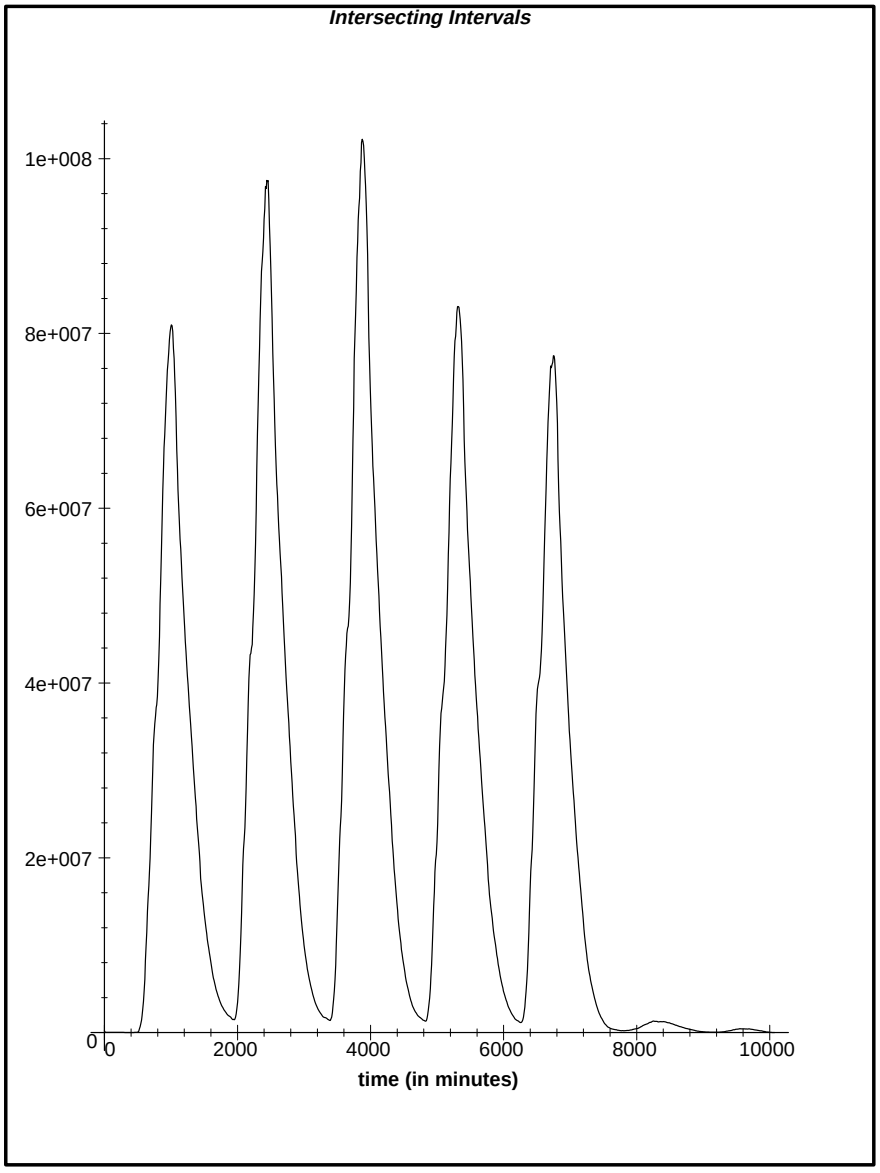


Figure 10.4: The profile of $R \bowtie_C R$ ("join 1").

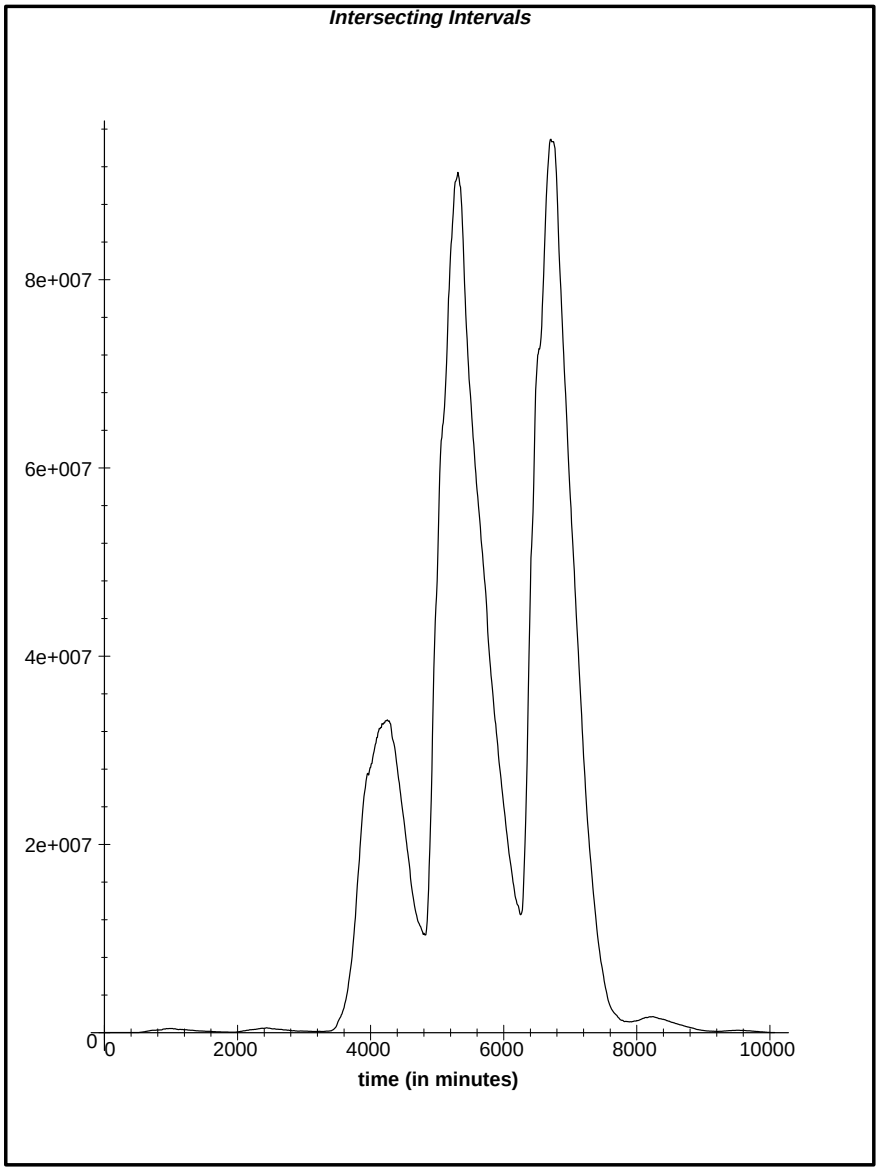


Figure 10.5: The profile of $R \bowtie_C Q$ (“join 2”).

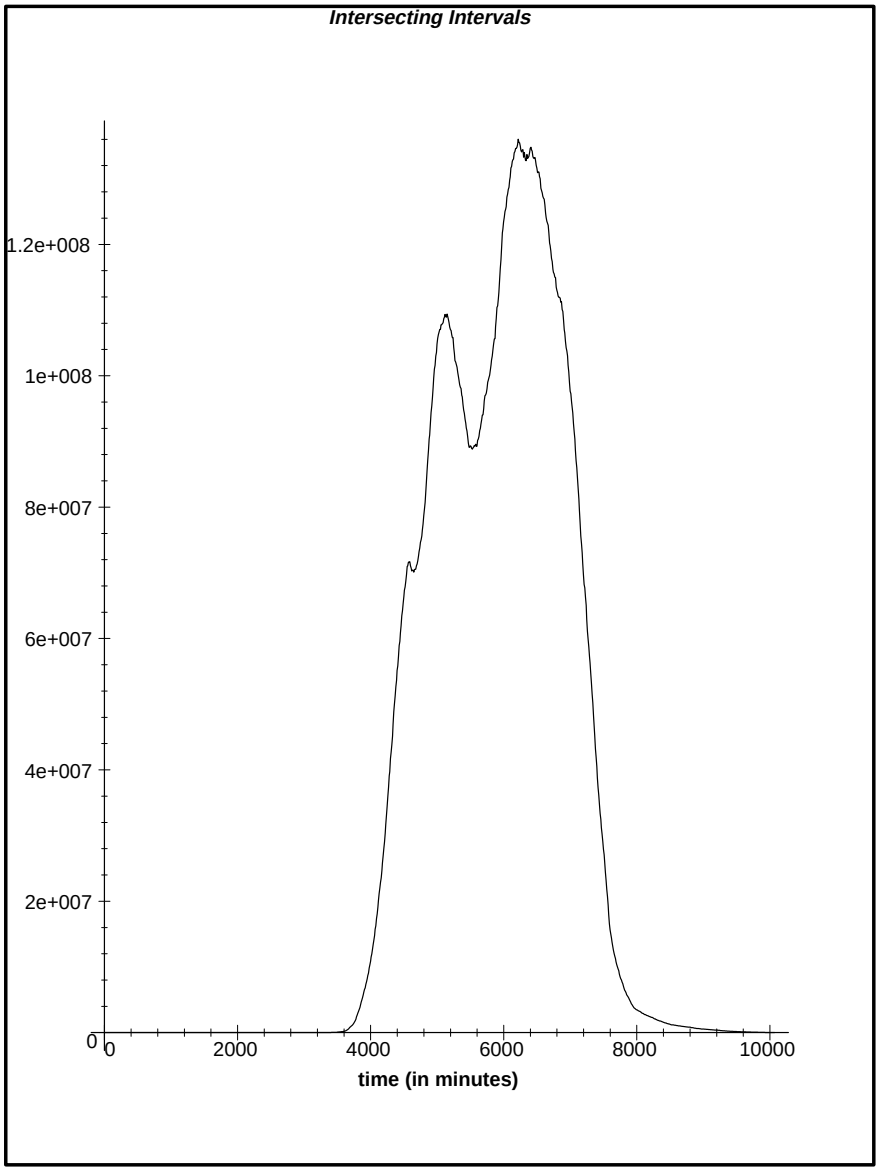


Figure 10.6: The profile of $Q \bowtie_C Q$ ("join 3").

Partitioning	$\mathcal{M}=4, \mathcal{N}=4$			$\mathcal{M}=1, \mathcal{N}=1$		
	$R \bowtie_C R$	$Q \bowtie_C Q$	$R \bowtie_C Q$	$R \bowtie_C R$	$Q \bowtie_C Q$	$R \bowtie_C Q$
Uniform Lifespan	3189	4327	3779	12783	19029	10322
Uniform Range	3207	4346	3755	12772	19053	10326
Uniform Startpoints Span	3225	4319	3773	12756	19019	10314
Basic Underflow	1490	2094	1884	11761	14900	8629
Basic Underflow with b/o	1695	2233	2085	11844	14852	8426
Prim. Underflow	1009	1194	1821	11706	14407	8868
Prim. Underflow with b/o	1120	1243	1634	11863	14392	8497
Basic Min.-Overlaps	1421	2001	2463	10542	14690	9199
Basic Min.-Overlaps with b/o	1502	2123	2780	10689	14714	8972
Prim. Min.-Overlaps	2298	1753	2243	11834	14621	8831
Prim. Min.-Overlaps with b/o	2457	1709	2619	12161	14599	9100

Table 10.3: The performance results (in sec.) for partitions with $m = 16$ fragments.

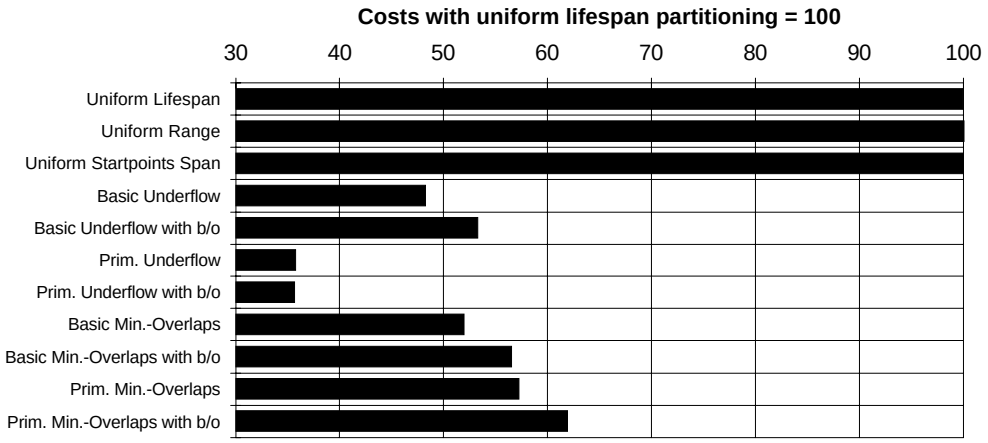


Figure 10.7: Performance result averages for the three joins on the parallel architecture.

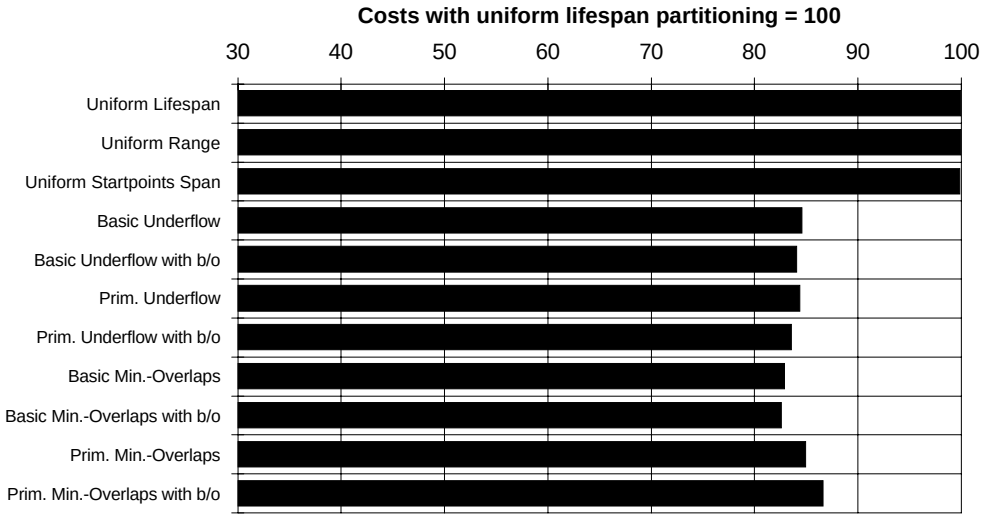


Figure 10.8: Performance result averages for the three joins on the single-processor architecture.

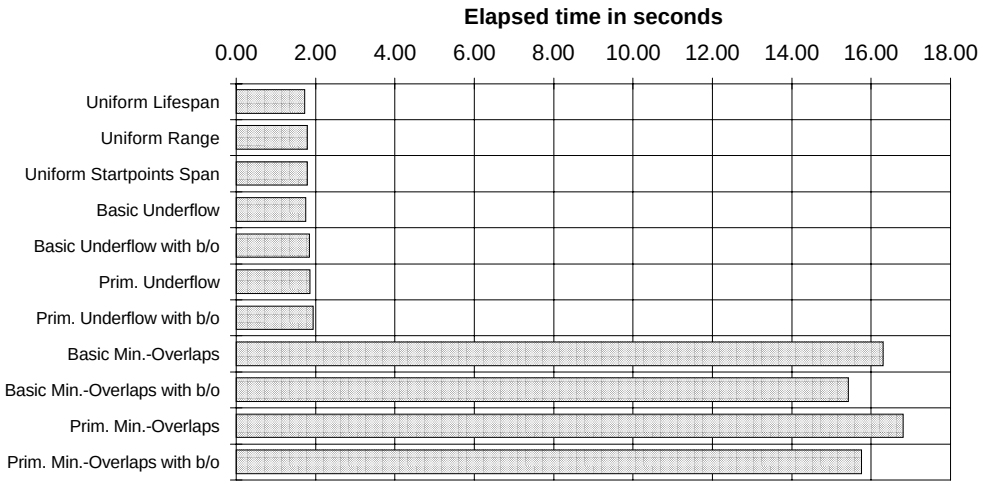


Figure 10.9: Average optimisation costs (in sec.) for all the experiments conducted in this section.

10.3 Dependency on m

In the experiment of previous section, the number m of fragments / partial joins was fixed to a value of 16 which matched the number of processors in the parallel architecture. Here, we want to find out whether this value was a good choice or whether higher values of m lead to better performances. In other words: we want to find out whether it is better to have a small number of relatively big partial joins or a large number of relatively small ones. There are no indications neither from the performance model (chapter 8) nor from the strategies themselves (chapter 9) whether one way or the other is better. Therefore, we have to rely on the observations made in the experiments. In order to reduce the complexity of the experiments and to concentrate on the main issues, we chose four of the eleven strategies on offer, namely the *uniform lifespan*, *basic underflow*, *primary underflow* and the *primary minimum-overlaps* strategies. The choice was led by the goal to select at least one member of each family and also those that are the most promising with respect to good performances.

Six experiments were conducted to test the dependency on m of these strategies, i.e. one per combination of join and hardware. The performance results are given in tables 10.4, 10.5 and 10.6 and visualised in figures 10.10 to 10.15.

First, we want to analyse the behaviour of the strategies on the parallel architecture. In all cases, the primary underflow strategy is a narrow winner before the primary minimum-overlaps strategy. Both strategies show relatively high costs on low values of m and perform better for higher values, with almost constant costs in the second half of the chart. The costs of the uniform lifespan strategy are between 2 and 3 times (join $R \bowtie_C R$), between 1.3 and 2 times (join $R \bowtie_C Q$) and between 2.2 and 3.5 times (join $Q \bowtie_C Q$) higher than those of the primary underflow strategy. In general, it also performs better for higher values of m. However, there are some exceptions to this rule such as a sharp rise from $m = 896$ to $m = 1024$ for the join $R \bowtie_C R$ or some significant shaking in the area between $m = 384$ and $m = 1024$ for the join $R \bowtie_C Q$. A much more irregular behaviour is shown by the basic underflow strategy. The exact shape of its cost function can only be explained by looking into details of the data. However, it seems that it performs best for lower values of m, which stands in contrast to the other three strategies. In general, it performs worse than the primary underflow and minimum-overlaps strategies.

Now, we turn our attention to the results for the single-processor hardware architecture. Here, the scene looks different: for lower values of m the uniform lifespan strategy has between 10% and 30% higher costs in comparison to the

primary underflow strategy, whereas for high values of m it performs between 1% and 4% better. This is a surprising result. It means in other words that the uniform lifespan strategy outperforms all other strategies if simply the value of m is chosen to be high enough. However, the advantage in comparison to other strategies is only very minor (see joins $R \bowtie_C R$ and $Q \bowtie_C Q$) or does not exist (see join $R \bowtie_C Q$). A further interesting result is that the primary minimum-overlaps strategy is better (joins $R \bowtie_C R$ and $R \bowtie_C Q$) or at least as good (join $Q \bowtie_C Q$) as the primary underflow strategy. This is due to the facts that (a) a good load balance between the partial joins is far less important on a single-processor machine in comparison to a parallel one and that (b) the advantage of having a reduced total number of overlapping intervals materialises in the single-processor case.

From the results in tables 10.4, 10.5 and 10.6 we can also derive some information about the *speed-up*. It is defined as the ratio between the processing times that are required for the same problem on a machine with n processors and on one with 1 processor. Ideally, the speed-up is n in this case. In realistic situations, however, such a speed-up is never achieved because of communication and synchronisation overheads. If we compare the respective lowest costs on the parallel and single-processor machines for the three joins, then we get speed-ups of 9.5 for $R \bowtie_C R$, 4.6 for $R \bowtie_C Q$ and 10.6 for $Q \bowtie_C Q$.

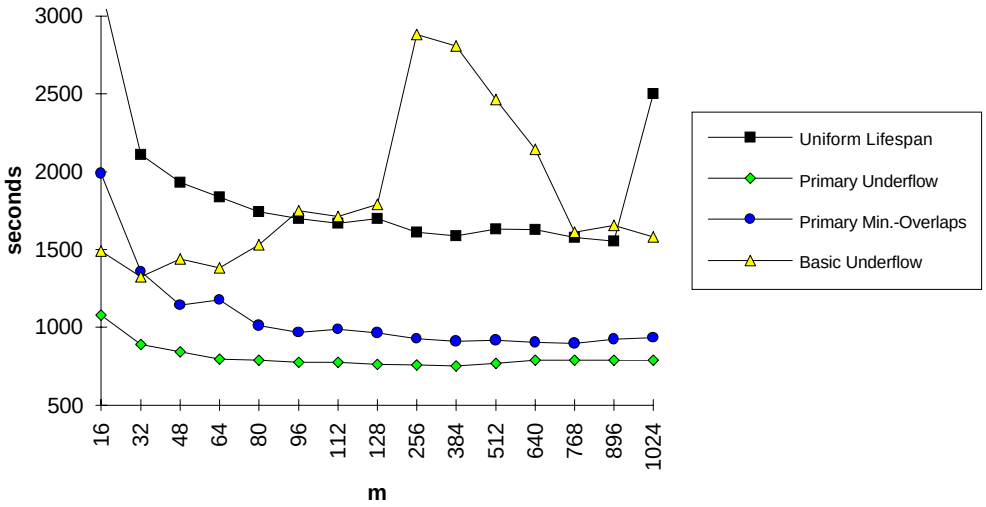


Figure 10.10: Dependency on m of the performance results for the join $R \bowtie_C R$ on a parallel architecture.

m	$\mathcal{M}=4, \mathcal{N}=4$				$\mathcal{M}=1, \mathcal{N}=1$			
	Uniform Lifespan	Primary Underflow	Primary Min.-O.	Basic Underflow	Uniform Lifespan	Primary Underflow	Primary Min.-O.	Basic Underflow
16	3189	1077	1988	1490	12783	11370	11286	11807
32	2111	891	1359	1326	10580	9589	9464	9852
48	1930	843	1141	1439	9675	8839	8750	9650
64	1837	797	1178	1384	9117	8458	8394	9437
80	1743	788	1011	1529	8783	8228	8177	9299
96	1699	777	967	1749	8559	8009	8029	9389
112	1668	776	986	1713	8395	7952	7929	9328
128	1700	763	963	1791	8241	7847	7855	9133
256	1613	757	926	2884	7194	7633	7618	9209
384	1587	752	910	2808	7258	7580	7569	8980
512	1630	770	916	2465	7310	7578	7568	8773
640	1629	788	903	2143	7351	7586	7399	8708
768	1577	788	898	1612	7379	7586	7493	8633
896	1554	788	923	1657	7416	7586	7543	8649
1024	2501	788	935	1582	7093	7586	7594	8666

Table 10.4: Performance results (in sec.) for the join $R \bowtie_C R$ depending on m.

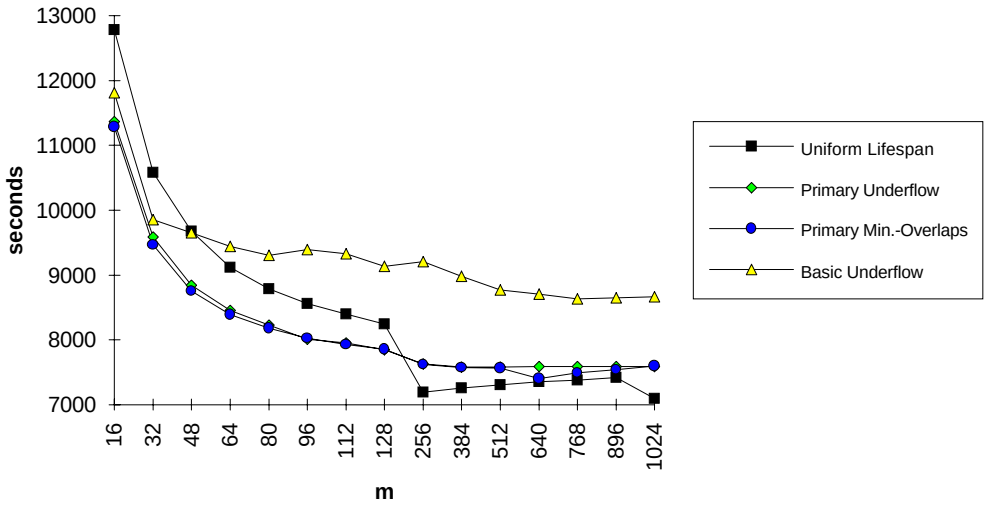


Figure 10.11: Dependency on m of the performance results for the join $R \bowtie_C R$ on a single-processor architecture.

m	$\mathcal{M}=4, \mathcal{N}=4$				$\mathcal{M}=1, \mathcal{N}=1$			
	Uniform Lifespan	Primary Underflow	Primary Min.-O.	Basic Underflow	Uniform Lifespan	Primary Underflow	Primary Min.-O.	Basic Underflow
16	3779	1901	2458	1884	10322	9248	9029	8629
32	2771	1454	1747	1647	7793	6966	6995	7229
48	2431	1339	1556	1531	6942	6395	6451	6693
64	2238	1295	1380	1468	6522	6111	6109	6320
80	2160	1253	1424	1476	6258	5930	5931	6235
96	2089	1224	1351	1575	6096	5814	5814	5612
112	2040	1209	1389	1697	5974	5740	5736	5616
128	1914	1202	1396	1613	5874	5679	5685	5625
256	1795	1169	1324	2398	5333	5501	5336	5688
384	1763	1147	1301	2274	5335	5376	5358	5704
512	1523	1170	1239	2544	5353	5402	5390	5760
640	1551	1174	1247	3251	5375	5436	5425	5827
768	1758	1171	1260	3432	5393	5472	5462	5863
896	1598	1171	1263	3051	5420	5503	5474	5886
1024	1699	1171	1263	2417	5462	5503	5474	5939

Table 10.5: Performance results (in sec.) for the join $R \bowtie_C Q$ depending on m.

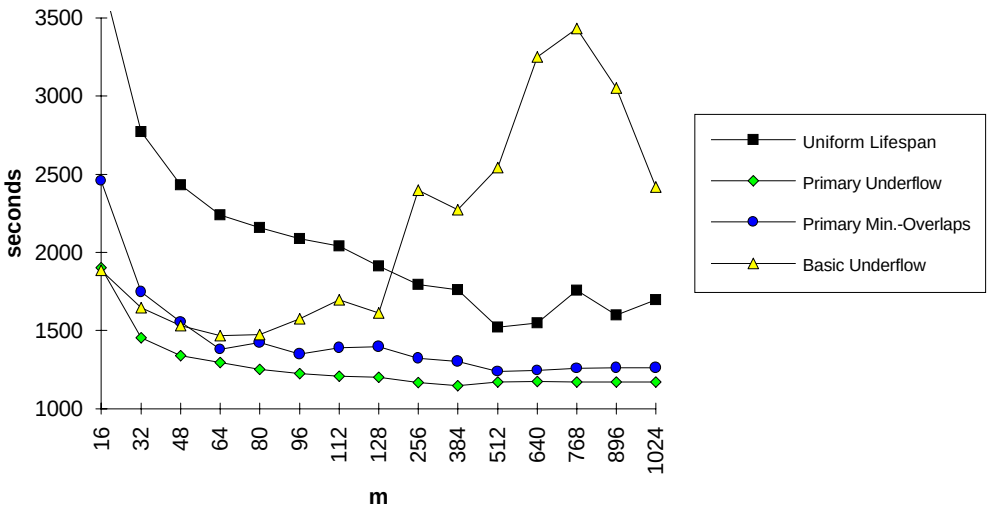


Figure 10.12: Dependency on m of the performance results for the join $R \bowtie_C Q$ on a parallel architecture.

m	$\mathcal{M}=4, \mathcal{N}=4$				$\mathcal{M}=1, \mathcal{N}=1$			
	Uniform Lifespan	Primary Underflow	Primary Min.-O.	Basic Underflow	Uniform Lifespan	Primary Underflow	Primary Min.-O.	Basic Underflow
16	4327	1243	1753	2122	19029	14551	14621	14739
32	3274	1058	1277	2165	14506	12153	12150	12878
48	2940	1001	1217	2281	12954	11389	11394	12330
64	2800	975	1111	2354	12189	10989	10986	12126
80	2683	964	1102	2590	11724	10773	10769	12076
96	2622	949	1060	2875	11413	10619	10616	12074
112	2575	936	1053	3111	11191	10514	10519	12030
128	2559	941	1047	4434	11010	10442	10441	11995
256	2451	938	1021	5912	10449	10214	10213	11973
384	2426	948	1013	6503	9964	10180	10179	12145
512	2447	963	1015	6522	9973	10195	10194	12062
640	2413	968	1025	6371	9992	10229	10228	12080
768	2430	997	1031	6117	10009	10272	10269	12095
896	2476	1009	1032	5626	10033	10323	10322	12001
1024	2252	1009	1048	2122	10072	10374	10376	12013

Table 10.6: Performance results (in sec.) for the join $Q \bowtie_C Q$ depending on m.

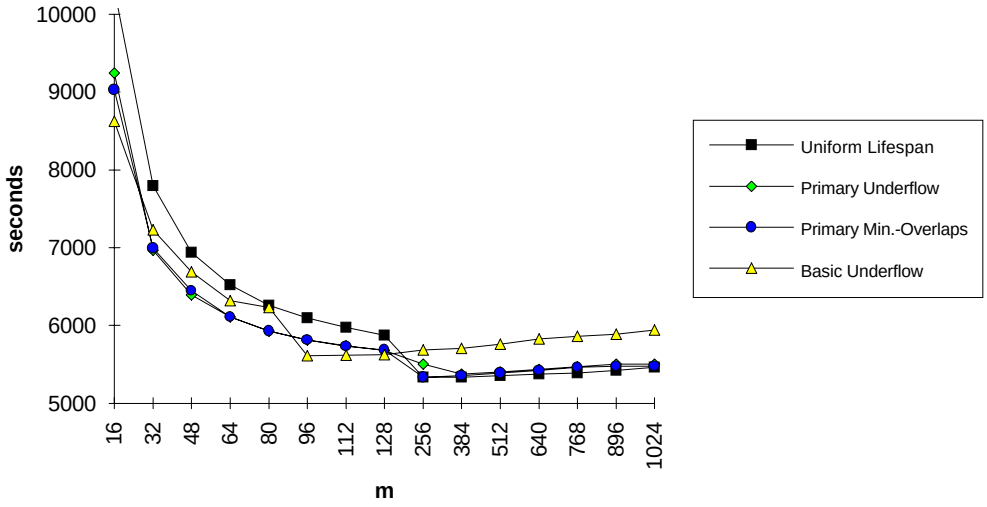


Figure 10.13: Dependency on m of the performance results for the join $R \bowtie_C Q$ on a single-processor architecture.

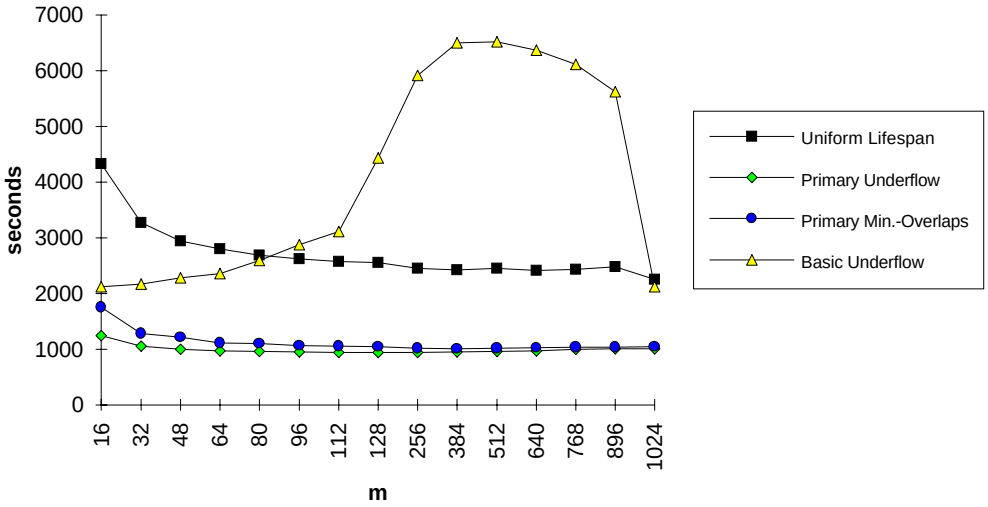


Figure 10.14: Dependency on m of the performance results for the join $Q \bowtie_C Q$ on a parallel architecture.

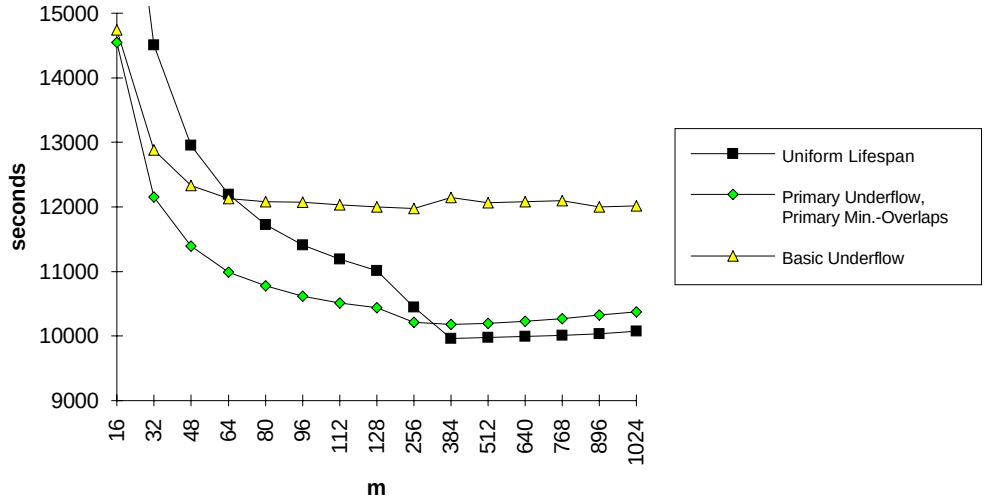


Figure 10.15: Dependency on m of the performance results for the join $Q \bowtie_C Q$ on a single-processor architecture.

10.4 Dependency on X_R and X_Q

In the previous section, we conducted experiments in which m was varied. Whereas m is an input parameter for the uniform strategies, it is an output parameter for the other strategies: it is implied by the value that we choose for X (in the case of the basic underflow and the basic minimum-overlaps strategies) or for X_R and X_Q (in the case of the primary underflow and the primary minimum-overlaps strategies). In section 10.3, we found out that high values for m usually result in a good performance when using the primary underflow or primary minimum-overlaps strategies. For practical purposes, this is a little bit vague and does not give a clear indication how the values for X_R and X_Q, i.e. the respective maximum sizes for the primary fragments R'_k and Q'_k, should be chosen. In this section, we want to overcome this deficit. We will try to empirically determine a rule that provides a good choice for X_R and X_Q. This is also an important piece of knowledge that we require for the experiments in the following sections.

X_R and X_Q are certainly parameters that depend on the numbers of tuples in R and Q, i.e. $|R|$ and $|Q|$. We set up a series of experiments in which we used a value Z to control X_R and X_Q using the following equations:

$$\begin{aligned} X_R &= |R| \cdot \frac{Z}{100} \\ X_Q &= |Q| \cdot \frac{Z}{100} \end{aligned}$$

Put in a different way: Z is the ratio (in percent) between the maximum number of tuples allowed per primary fragment and the number of tuples in the relation. For example $Z = 5$ means that X_R and X_Q are 5% of the value of $|R|$ and $|Q|$ respectively.

Several experiments were conducted, using 10, 9, ... , 1, 0.75, 0.5, 0.25, 0.2 as values for Z. Table 10.7 shows the performances for the three joins using the primary underflow strategy on the parallel architecture. The experiment was run for R and Q and then again for samples of 40000 tuples of these two relations. From this variation we hope to see whether a good choice of a value for Z is really independent of $|R|$ and $|Q|$. Figures 10.16 and 10.17 visualise the data of table 10.7. In all cases, a value of $Z = 0.75$ provides the best performance.

We then moved on to see whether a similarly clear result can be obtained for the primary minimum-overlaps strategy. Table 10.8 gives the performances and figures 10.18 and 10.19 show them graphically. Here, we find that Z-values

Z	$\|R\| = \|Q\| = 121728$			$\|R\| = \|Q\| = 40000$		
	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)
10	1771	1823	2140	193	198	231
9	1681	1675	1876	183	181	202
8	1433	1601	1606	156	173	172
7	1203	1495	1356	132	161	146
6	1249	1657	1377	137	179	149
5	1455	1477	1583	160	160	170
4	1146	1435	1367	126	156	147
3	1051	1178	1044	115	129	113
2	836	1260	964	92	138	105
1	753	1174	930	85	131	103
0.75	708	1137	896	80	127	100
0.50	775	1147	949	88	131	108
0.25	766	1158	942	94	141	113
0.20	782	1165	956	96	144	118

Table 10.7: Performance results (in sec.) depending on Z for the three joins and the primary underflow strategy on the parallel architecture.

between 0.2 and 0.75 provide best performances. $Z = 0.2$ is the best choice if near-optimal performances are to be achieved for all cases.

Next, we looked at the performances that are achieved on a single-processor machine. First, the primary underflow strategy was tested. The results are shown in table 10.9 and visualised by the graphs in figures 10.20 and 10.21. Here, we find the best performances for Z-values in the range 0.2 to 1. If we consider all performances for these Z-values then $Z = 0.5$ is best because it near-optimal performances in all cases.

Finally, the primary minimum-overlaps strategy was investigated. The results are shown in table 10.10 and visualised by the graphs in figures 10.22 and 10.23. Similar to the primary underflow case, it is the Z-values between 0.2 and 1 that perform best with $Z = 0.5$ achieving near-optimal performances in all cases.

In summary, we conclude that values for $Z = 0.75$ on the parallel architecture and $Z = 0.5$ on the single-processor machine seem to be good choices in general. We will use these values to determine X_R and X_Q for the primary underflow and primary minimum-overlaps strategies in the remaining experiments.

Z	$\|R\| = \|Q\| = 121728$			$\|R\| = \|Q\| = 40000$		
	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)
10	2523	2576	3164	274	276	336
9	2367	2251	2780	263	255	295
8	2523	2124	2343	275	207	250
7	2082	2433	1948	229	261	208
6	2150	2302	1964	236	248	212
5	1871	1773	1970	205	192	211
4	1759	1640	1694	192	198	182
3	1454	1545	1327	159	167	140
2	1389	1491	1201	153	157	141
1	1190	1373	1097	133	150	121
0.75	1148	1337	1114	126	149	116
0.50	985	1324	1058	114	146	120
0.25	922	1256	1014	111	151	121
0.20	912	1225	1025	114	148	125

Table 10.8: Performance results (in sec.) depending on Z for the three joins and the primary minimum-overlaps strategy on the parallel architecture.

Z	$\|R\| = \|Q\| = 121728$			$\|R\| = \|Q\| = 40000$		
	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)
10	13352	9211	17218	1452	999	1848
9	14308	8704	16337	1553	941	1749
8	12865	8023	15571	1402	868	1666
7	12471	7845	14892	1356	849	1597
6	11371	7427	14163	1239	804	1514
5	10631	7084	13471	1155	768	1443
4	10148	6691	12744	1110	724	1364
3	9406	6291	12004	1022	682	1285
2	8753	5947	11284	956	647	1208
1	8014	5604	10578	880	591	1138
0.75	7858	5533	10410	867	593	1123
0.50	7665	5362	10262	854	602	1114
0.25	7580	5444	10182	861	635	1128
0.20	7578	5496	10199	871	653	1142

Table 10.9: Performance results (in sec.) depending on Z for the three joins and the primary underflow strategy on the single-processor architecture.

Z	$\|R\| = \|Q\| = 121728$			$\|R\| = \|Q\| = 40000$		
	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)
10	12833	9216	17043	1394	995	1829
9	12482	8702	16242	1358	949	1736
8	12310	8444	15462	1336	895	1656
7	11497	7782	14840	1250	843	1589
6	11084	7484	14095	1203	807	1506
5	10381	7073	13426	1119	765	1435
4	9781	6769	12691	1058	729	1360
3	9296	6357	11995	1014	689	1284
2	8611	5967	11281	941	652	1209
1	7990	5626	10578	778	587	1138
0.75	7826	5335	10410	864	589	1123
0.50	7668	5342	10260	802	597	1114
0.25	7567	5420	10180	834	628	1128
0.20	7569	5467	10195	848	645	1141

Table 10.10: Performance results (in sec.) depending on Z for the three joins and the primary minimum-overlaps strategy on the single-processor architecture.

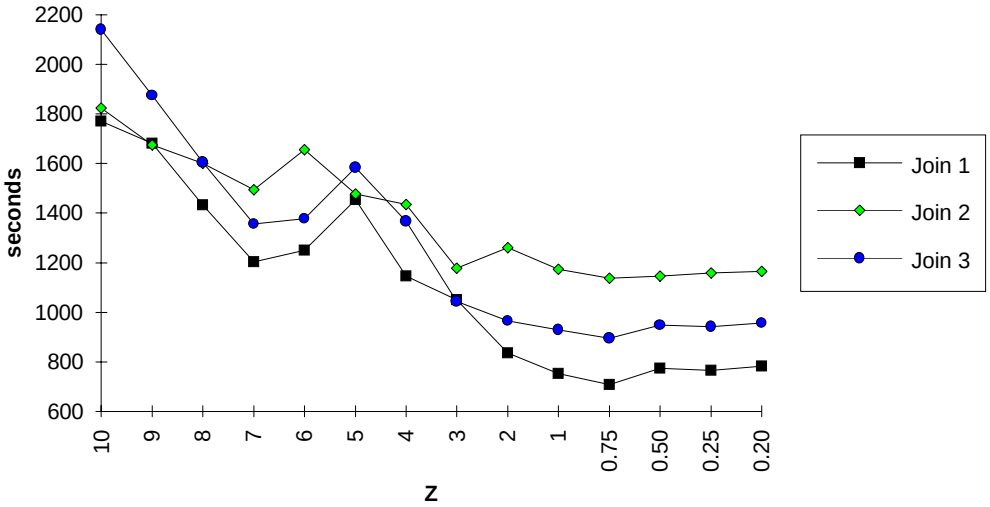


Figure 10.16: Dependency on Z of the performance results for $|R| = |Q| = 121728$ and the primary underflow strategy on the parallel architecture.

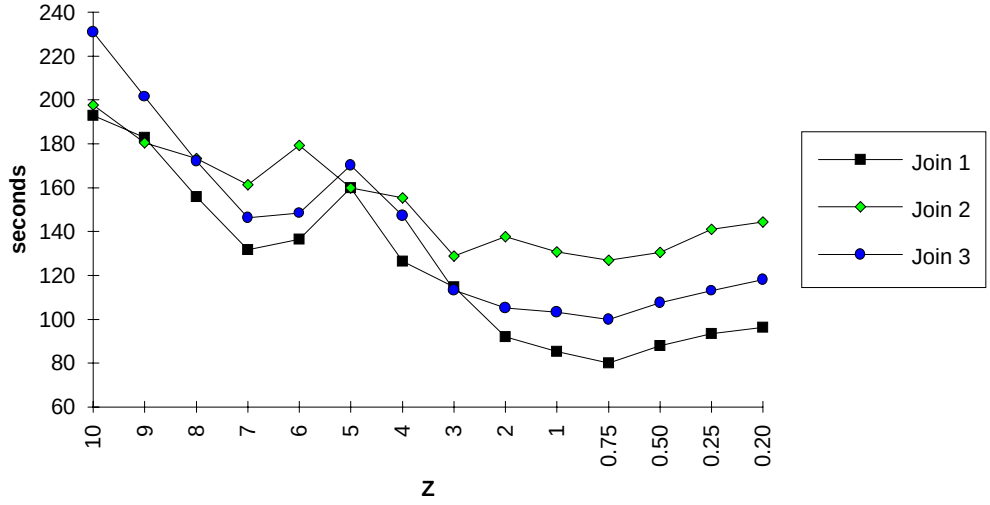


Figure 10.17: Dependency on Z of the performance results for $|R| = |Q| = 40000$ and the primary underflow strategy on the parallel architecture.

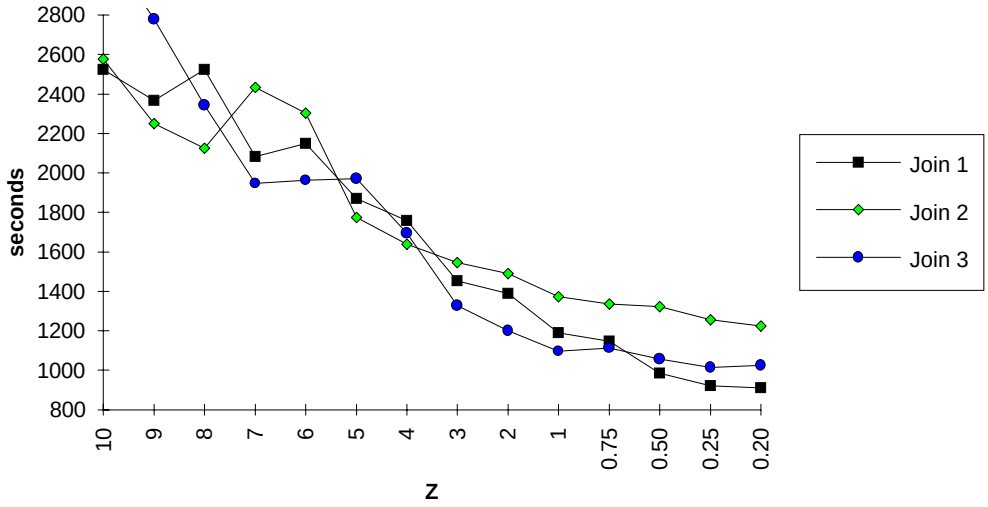


Figure 10.18: Dependency on Z of the performance results for $|R| = |Q| = 121728$ and the primary minimum-overlaps strategy on the parallel architecture.

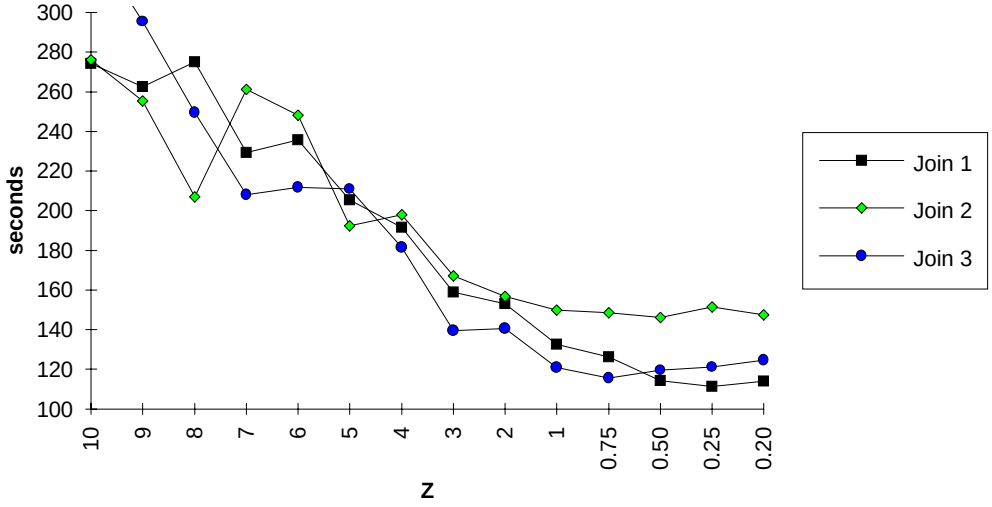


Figure 10.19: Dependency on Z of the performance results for $|R| = |Q| = 40000$ and the primary minimum-overlaps strategy on the parallel architecture.

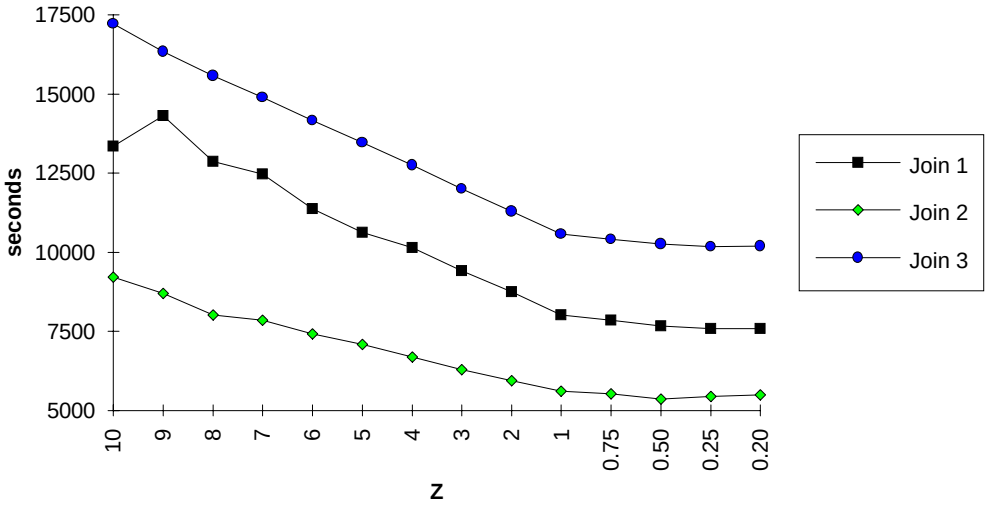


Figure 10.20: Dependency on Z of the performance results for $|R| = |Q| = 121728$ and the primary underflow strategy on the single-processor machine.

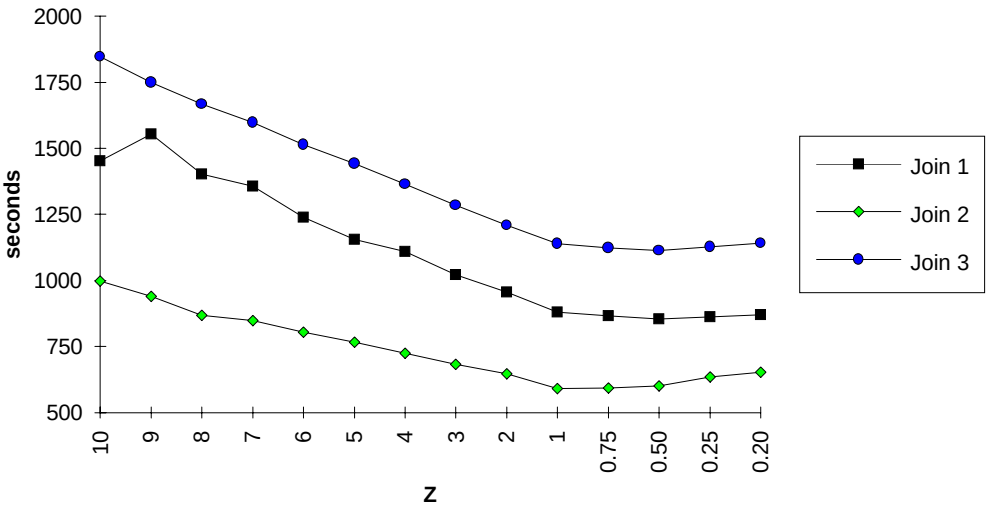


Figure 10.21: Dependency on Z of the performance results for $|R| = |Q| = 40000$ and the primary underflow strategy on the single-processor machine.

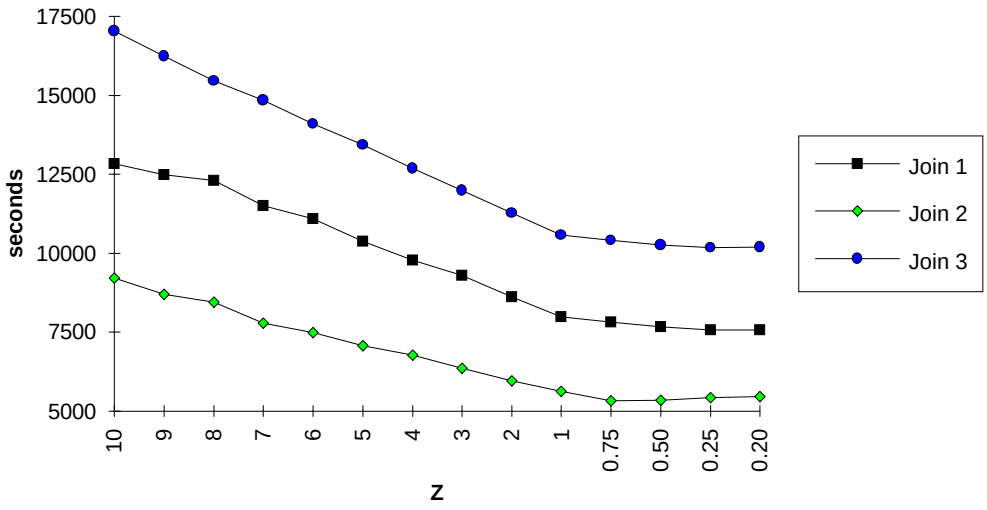


Figure 10.22: Dependency on Z of the performance results for $|R| = |Q| = 121728$ and the primary minimum-overlaps strategy on the single-processor machine.

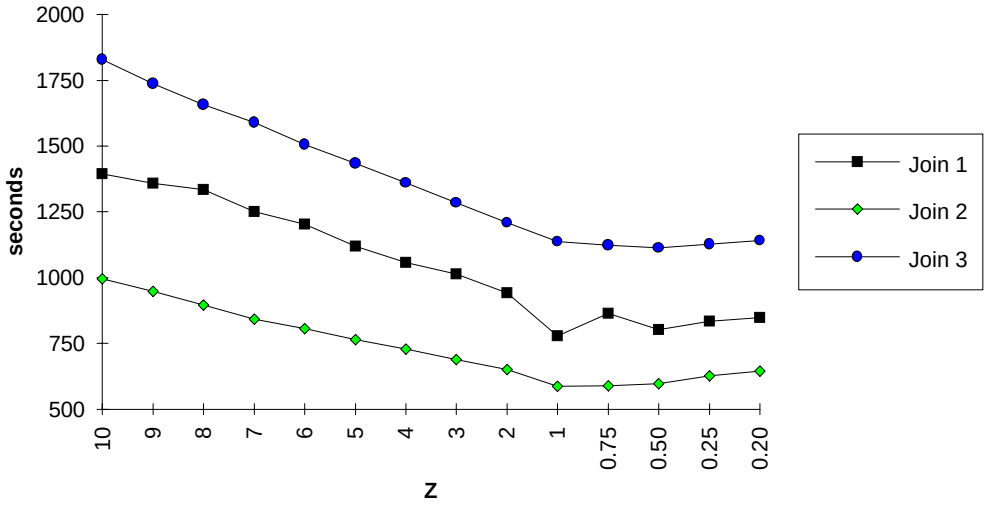


Figure 10.23: Dependency on Z of the performance results for $|R| = |Q| = 40000$ and the primary minimum-overlaps strategy on the single-processor machine.

10.5 Dependency on τ

We now want to look at the influence of a parameter that is imposed by the data, namely the average length τ of the intervals of a temporal relation. Obviously, the amount of overlapping intervals grows if τ is increased. This can influence the performance levels that result from the various strategies. In this section, we want to investigate the relationship between τ and the performance.

For that purpose, we used our base relations R and Q which both have $\tau = 300$. We applied the procedure *change_lengths* (see appendix C) to derive relations R_τ and Q_τ with average interval length τ respectively. Values for τ were 200, 400, 600, 800, 1000 and 1200. The corresponding profiles $i_{R_\tau}(t)$ and $i_{Q_\tau}(t)$ are listed in appendix D. With increasing τ the profiles become smoother, showing less significant peeks than for low values of τ.

In the experiments, we simulated the performances of the joins $R_\tau \bowtie_C R_\tau$, $R_\tau \bowtie_C Q_\tau$ and $Q_\tau \bowtie_C Q_\tau$ for the values of τ that are listed above. In order to restrict the combinatorical complexity, we concentrated on one partitioning strategy per family, namely the uniform lifespan, the primary underflow and the primary minimum-overlaps strategies. All these strategies have been the best performing members of their families in most cases so far. Tables 10.11 and 10.12 show the performance results (in sec.) for the parallel and for the single-processor architecture, respectively. As before, it is difficult to see the effects by simply looking at the absolute numbers. Therefore we have visualised the results in figures 10.24, 10.25 and 10.26 for the parallel architecture and in figures 10.27, 10.28 and 10.29 for the single-processor architecture.

In the parallel case, we can recognise significant differences depending on the join:

- For the joins $R_\tau \bowtie_C R_\tau$ – where the profiles of both participating relations are periodic – we find that the primary underflow strategy performs best for low τ whereas the primary minimum-overlaps strategy is the clear winner for high values of τ. Obviously, the longer the intervals become the more overlaps occur and the more relevant the problem of overlaps becomes. This seems to favour the primary minimum-overlaps strategy for high values of τ.

- For the joins $R_\tau \bowtie_C Q_\tau$ – where the profile of one relation is periodic whereas the other one's profile is non-periodic – we find that the primary

minimum-overlaps strategy performs best or at least close to the best for all values of τ. The reason behind this is the same as in the previous case.

- For the joins $Q_\tau \bowtie_C Q_\tau$ – where the profiles of both participating relations are non-periodic – we find that the primary underflow strategy performs best for all τ. Its advantage over the other strategies seems to increase for a growing τ. This stands in contrast to the two other joins and is due to the fact that the Q_τ's profiles (see figures D.7 to D.12 in appendix D) offer less and less opportunities to set a breakpoint in a valley with an increasing τ. The primary minimum-overlaps strategy, however, usually tries to take advantage of such opportunities which it cannot in this particular case. It is therefore that the primary underflow strategy, which focuses on a good load balance, proves to provide better performances.

We then took the average for the three strategies over the three joins per value of τ. The times were normalised in the same way as described in section 10.2 in order to guarantee a fair comparison. Figure 10.30 shows the normalised performances. We note that these averages suggest that the primary underflow strategy performs best in most cases. This stands in contrast to the more detailed analysis above. However, it is possible to recognise a trend that the performances of primary underflow and primary minimum-overlaps partitioning approach the one of uniform lifespan partitioning. This is not surprising when we consider the profiles of the relations that were used in the experiments (see appendix D): with an increasing τ they loose their respective periodic and non-periodic characteristics and approach a profile of for uniform data which has a constant profile, i.e. $i_R(t)$ and $i_Q(t)$ would be constant.

Now, we turn to the results that were obtained for the single-processor architecture. These are shown in table 10.12 and in figures 10.27 to 10.29. In contrast to the parallel case, we cannot observe any advantages for a particular strategy for low or high values of τ. The normalised averages in figure 10.31 show a slow convergence with an increasing τ.

In summary, it is fair to say that the average interval length τ is a significant parameter in the case of a parallel architecture whereas it is almost neglectable on single-processor machines. For parallel join processing, however, we conclude that low values of τ favour the primary underflow strategy, whereas high τ cause the periodicity or non-periodicity of the participating relations' profiles to be a distinguishing factor. The presence of periodic profiles suggests that primary minimum-overlaps partitioning is the a good choice whereas the

absence of such profiles indicates advantages for the primary underflow strategy.

τ	$R_\tau \bowtie_C R_\tau$			$R_\tau \bowtie_C Q_\tau$			$Q_\tau \bowtie_C Q_\tau$		
	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps
200	1145	559	633	1269	817	814	1692	688	716
400	1914	974	1550	1985	1451	1617	3188	1241	1504
600	2286	1161	1865	2939	1850	1934	4232	1778	2476
800	2513	2513	2357	3214	3214	2415	5750	2237	3153
1000	2992	2992	2658	3598	3598	3138	6954	2659	4377
1200	3817	3817	3043	4071	4071	4386	7073	3074	5574

Table 10.11: Dependency on τ of the performance results (in sec.) for the three joins on the parallel architecture.

τ	$R_\tau \bowtie_C R_\tau$			$R_\tau \bowtie_C Q_\tau$			$Q_\tau \bowtie_C Q_\tau$		
	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps
200	5331	5905	5892	3564	3785	3793	6745	7241	7238
400	9420	9671	9643	7076	7292	7063	13045	13534	13530
600	12526	12762	12736	10465	10683	10426	18885	19369	19369
800	15440	15440	15618	13782	13782	13726	24246	24722	23942
1000	18429	18429	18558	16999	16999	16908	29075	29550	28412
1200	21661	21661	21743	20375	20375	20371	33801	33940	33926

Table 10.12: Dependency on τ of the performance results (in sec.) for the three joins on the single-processor architecture.

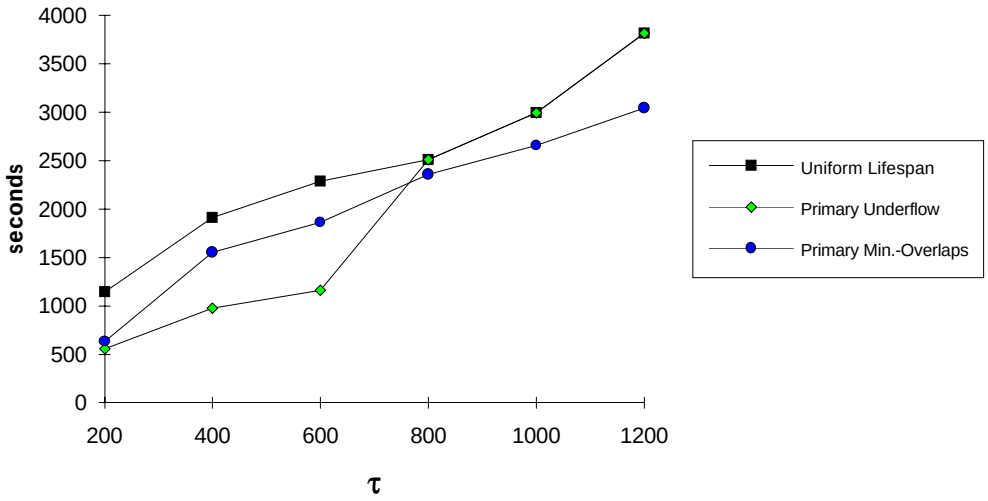


Figure 10.24: Performances for the joins $R_\tau \bowtie_C R_\tau$ on the parallel architecture.

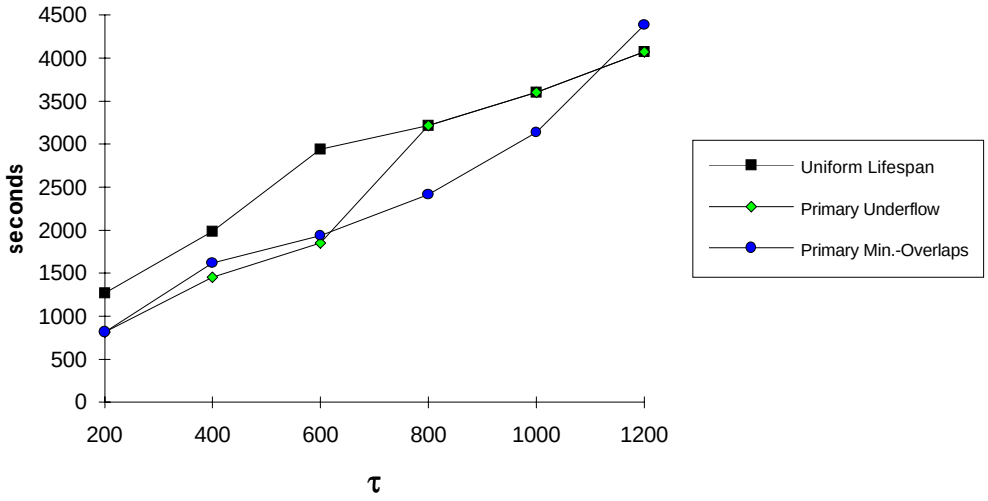


Figure 10.25: Performances for the joins $R_\tau \bowtie_C Q_\tau$ on the parallel architecture.

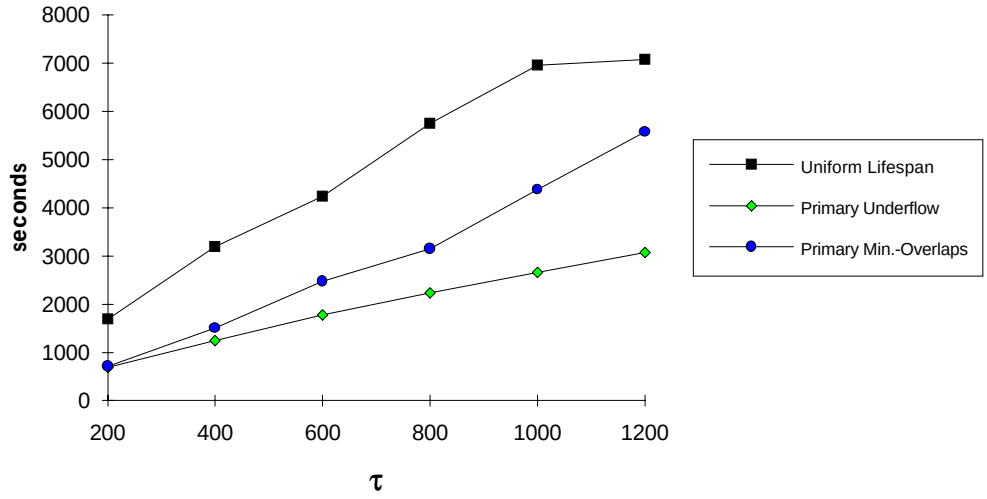


Figure 10.26: Performances for the joins $Q_\tau \bowtie_C Q_\tau$ on the parallel architecture.

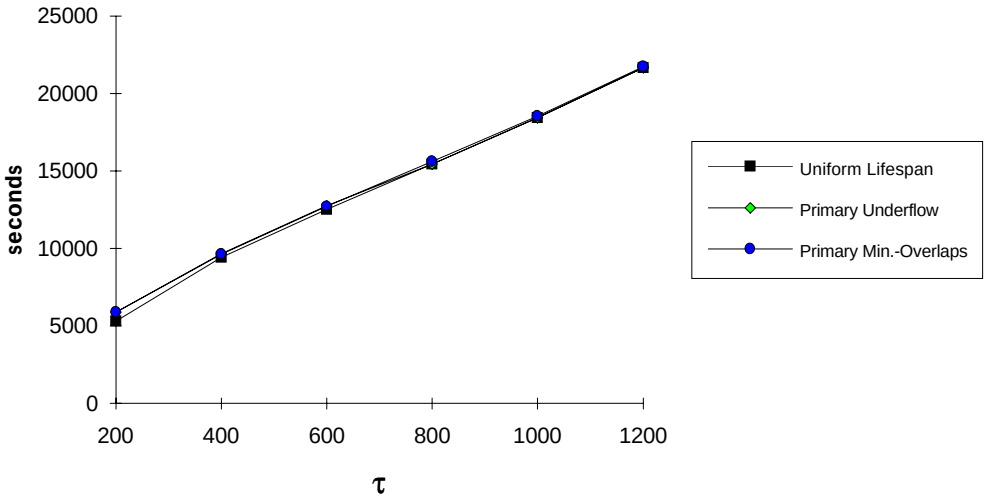


Figure 10.27: Performances for the joins $R_\tau \bowtie_C R_\tau$ on the single-processor architecture.

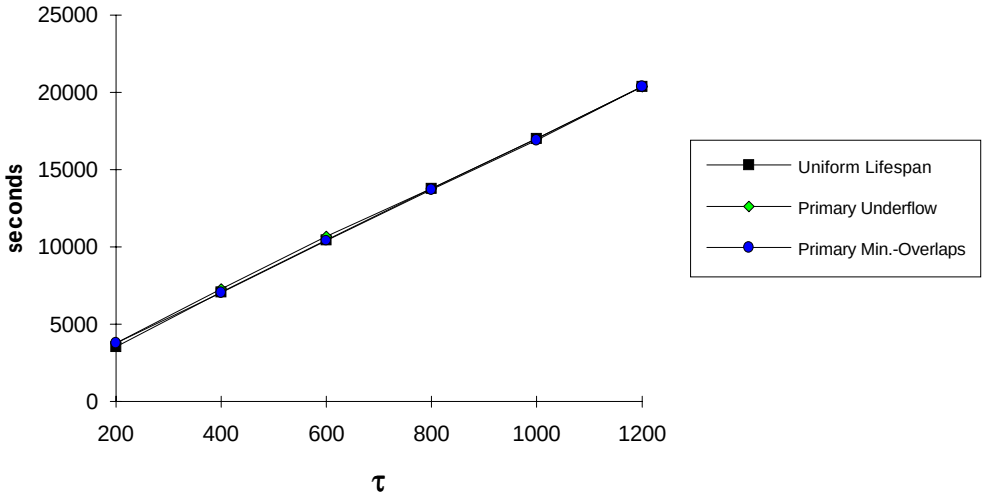


Figure 10.28: Performances for the joins $R_\tau \bowtie_C Q_\tau$ on the single-processor architecture.

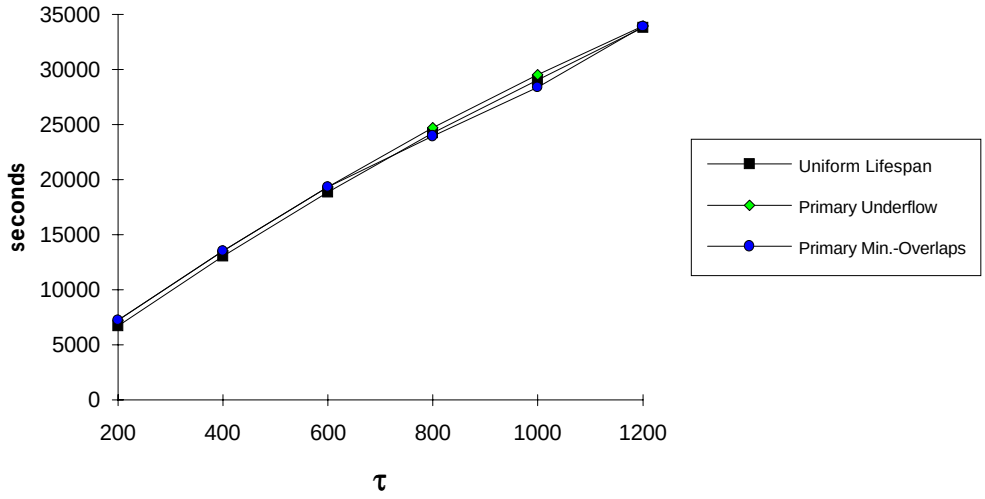


Figure 10.29: Performances for the joins $Q_\tau \bowtie_C Q_\tau$ on the single-processor architecture.

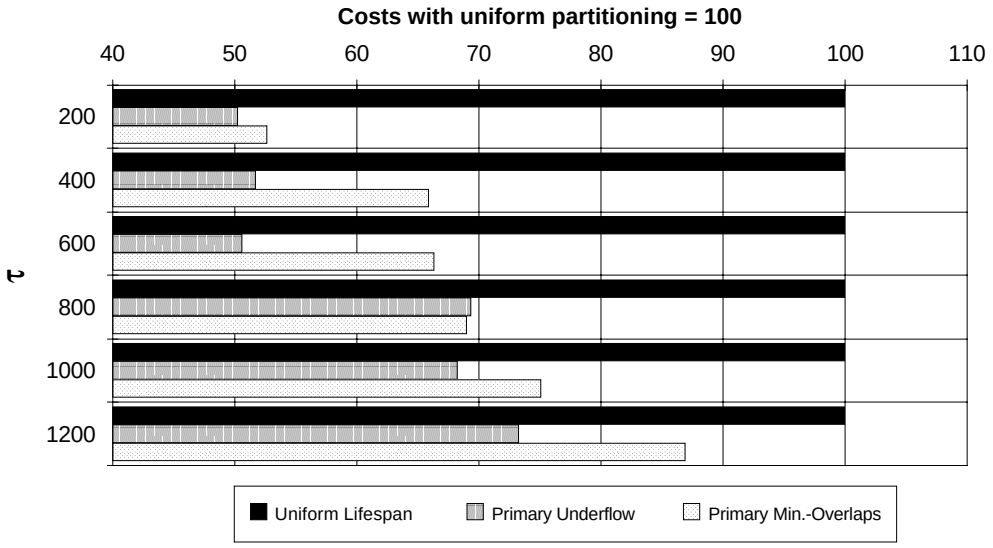


Figure 10.30: Comparison between the performances of the three strategies on a parallel architecture with a varying τ.

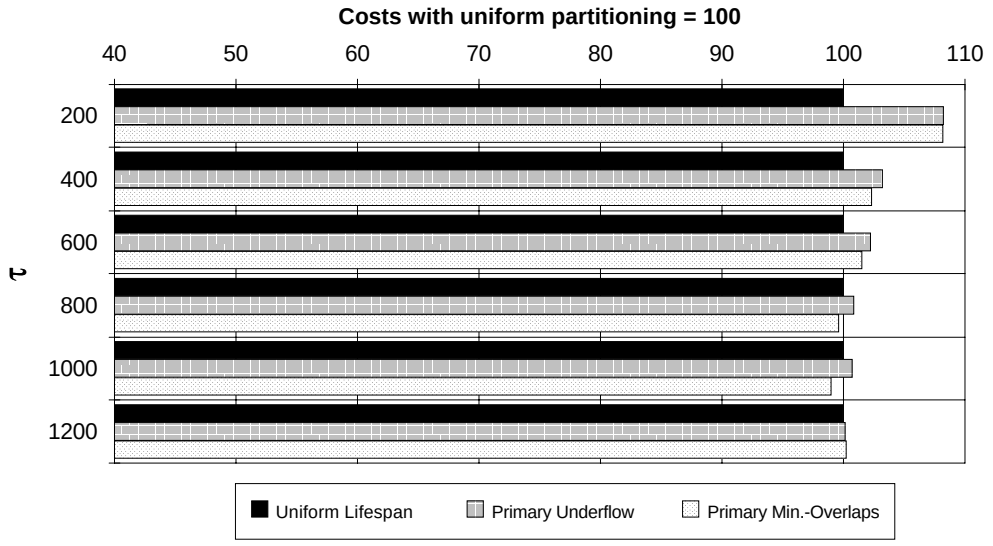


Figure 10.31: Comparison between the performances of the three strategies on a single-processor architecture with a varying τ.

10.6 Dependency on $|R|$ and $|Q|$

In this section, we want to find out whether the sizes of the participating relations can make a difference to the optimisation decision on the most suitable partitioning strategy. The experiments were set up as in the previous section but this time τ remained constant at its original value of 300. Here, we varied the number of tuples within the participating relations, i.e. $|R|$ and $|Q|$. The relations were created by randomly picking tuples from the original sets which have 121728 tuples each. In this section's experiments the sample sizes are 20000, 40000, 60000, 80000 and 100000.

Table 10.13 shows the performance results for the three joins on the parallel architecture; table 10.14 shows the corresponding figures for the single-processor machine. There are no big surprises within these numbers. Figures 10.32 and 10.33 therefore show the averages for the three joins respectively. The quadratic time complexity of the underlying join algorithm can be easily recognised on both machines. In figures 10.34 and 10.35 the normalised results are shown. Here, we do not see any surprising effects either. In total, we can conclude that the relation sizes have no impact on the choice of the best partitioning strategy.

As a "side-result" of these experiments we note that the numbers for the skewed data (table 10.13) are quite different from those that were obtained under the assumption of uniformity (see table 8.10 on page 211): for example, for $|R|, |Q| = 100000$ we got 386.5 seconds with uniform data and 538, 753 and 545 seconds with the primary underflow strategy with the skewed data. This difference underlines how important it is to consider data skew and to look at techniques that can cope with this effect. Especially the results in table 10.13 clearly show how badly the uniform strategy performs as it does not consider any characteristics of the underlying data apart from the lifespan (or range or startpoints span).

$\|R\|, \|Q\|$	$R \bowtie_C R$			$R \bowtie_C Q$			$Q \bowtie_C Q$		
	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps
20000	47	22	33	52	34	39	72	27	33
40000	180	80	126	197	127	149	269	100	116
60000	395	177	284	436	280	335	595	221	272
80000	813	366	684	610	419	423	582	239	253
100000	1175	538	914	1170	753	888	1486	545	661

Table 10.13: Dependency on $|R|$, $|Q|$ of the performance results (in sec.) for the three joins on the parallel architecture.

$\|R\|, \|Q\|$	$R \bowtie_C R$			$R \bowtie_C Q$			$Q \bowtie_C Q$		
	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps
20000	211	221	221	157	155	154	278	287	287
40000	811	867	864	594	593	589	1079	1123	1123
60000	1798	1933	1927	1319	1320	1313	2413	2517	2517
80000	3825	4062	4078	2153	2164	2242	2353	2543	2543
100000	5463	5867	5853	3683	3695	3679	5990	6290	6289

Table 10.14: Dependency on $|R|$, $|Q|$ of the performance results (in sec.) for the three joins on the single-processor architecture.

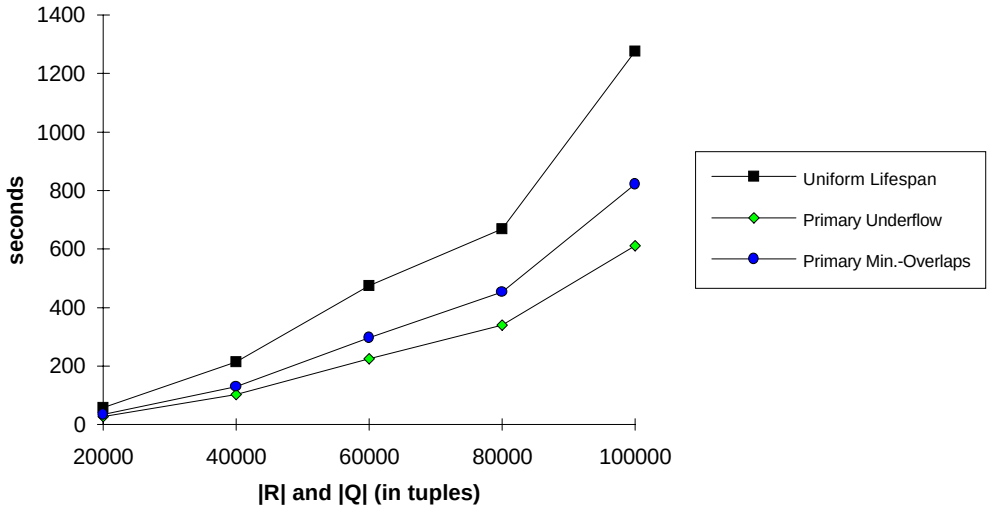


Figure 10.32: Performance averages for the three joins on the parallel architecture for varying $|R|$ and $|Q|$.

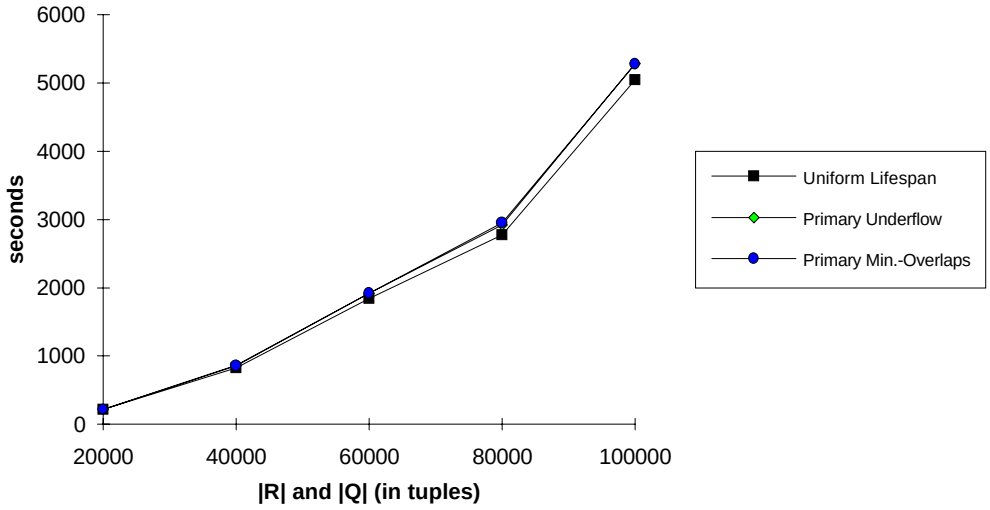


Figure 10.33: Performance averages for the three joins on the single-processor architecture for varying $|R|$ and $|Q|$.

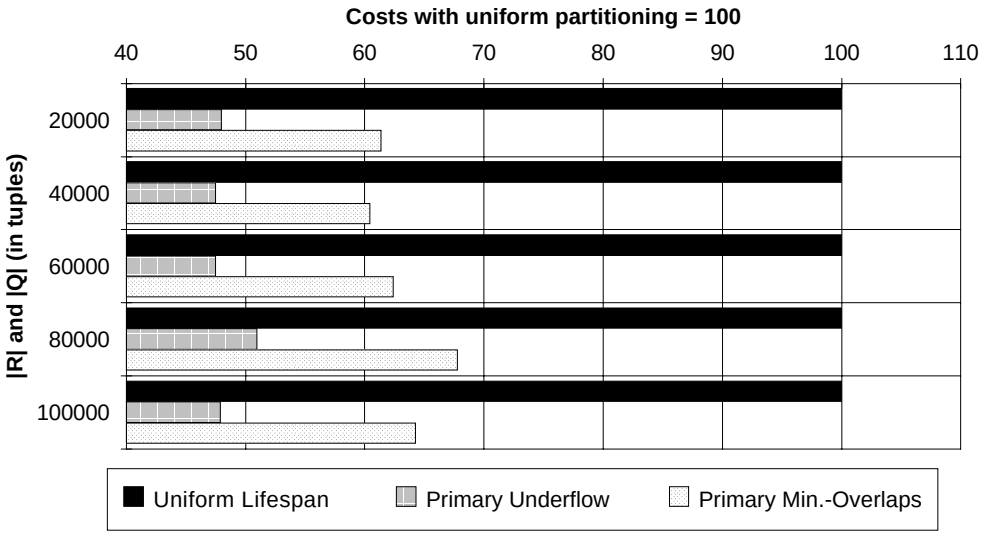


Figure 10.34: Comparison between the performances of the three strategies on a parallel architecture for varying $|R|$ and $|Q|$.

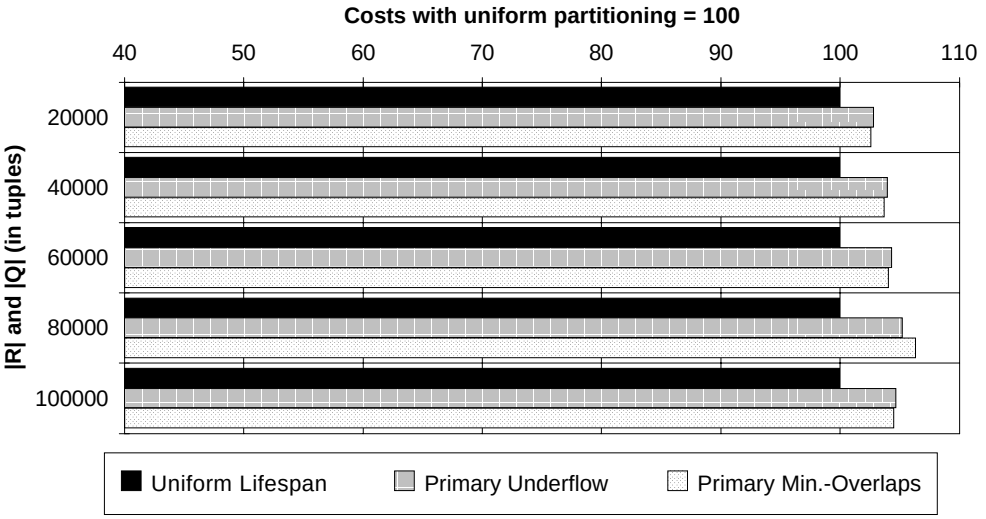


Figure 10.35: Comparison between the performances of the three strategies on a single-processor architecture for varying $|R|$ and $|Q|$.

10.7 The Architectural Influence

In experiment 1 of section 8.5, we already tried to find out which parallel architecture, i.e. which type and which mixture of SMP-nodes, would be most appropriate. Here, we repeat this experiment. This time, however, the workload will not be uniform but skewed. As in section 8.5, the $\mathcal{M}/\mathcal{N}$ combinations 1/16, 2/8, 4/4, 8/2 and 16/1 will be investigated, i.e. there will be a total of $\mathcal{M} \cdot \mathcal{N} = 16$ processors in all cases. As before, we will run the three joins $R \bowtie_C R$, $R \bowtie_C Q$ and $Q \bowtie_C Q$ on these architectures using the uniform lifespan (with $m = 384$), the primary underflow (with $X_R = 0.0075 \cdot |R|$, $X_Q = 0.0075 \cdot |Q|$) and the primary minimum-overlaps (with $X_R = 0.0075 \cdot |R|$, $X_Q = 0.0075 \cdot |Q|$) strategies.

Table 10.15 shows the results of these experiments. The performances are visualised in figures 10.36, 10.37 and 10.38. Overall, the shapes of the cost graphs are similar to the one of figure 8.15 (page 212). However, there are three effects which are slightly out of line:

- For the joins $R \bowtie_C R$ and $Q \bowtie_C Q$ the combination $\mathcal{M}$ / $\mathcal{N}$ seems to make a difference, at least for underflow and minimum-overlaps partitioning. 4/4, 8/2 and 16/1 are the favourable combinations, causing only around 40% of the costs in most cases and in comparison to the 1/16 architecture. However, this is higher than in experiment 1 of section 8.5 where the 4/4, 8/2 and 16/1 architectures had only around 26% of the costs of the 1/16 architecture. This proves again that the somewhat unrealistic assumption of uniformity presented a distorted picture of the figures that can be expected for real applications.

- For the join $R \bowtie_C Q$, the performance advantage of the 4/4, 8/2 and 16/1 combinations is between 10% and 20% for all strategies in comparison to the 1/16 architecture. This is rather low when compared to the 60% gains for the other joins.

- For the join $Q \bowtie_C Q$, the performance results for uniform partitioning are almost the same between the architectures. On the other hand, the results change a lot for primary underflow and primary minimum-overlaps partitioning.

The effects that we have observed here can be explained by looking into the components that contribute to the costs. These are shown in tables 10.16, 10.17 and 10.18 respectively. As an example, the numbers for the primary underflow

strategy were visualised in figures 10.39, 10.40 and 10.41. For the joins $R \bowtie_C R$ and $Q \bowtie_C Q$, we find that the memory access costs dominate the processing of the subjoins for the 1/16 and 2/8 architectures whereas the CPU costs dominate in the case of the 4/4, 8/2 and 16/1 architectures. In the case of the join $R \bowtie_C Q$ it is the CPU costs that dominate in most situations. Therefore, the mixture between closely and loosely coupled processors is not as significant as for the other joins. This can also be seen in the case of uniform lifespan partitioning for the join $Q \bowtie_C Q$. In contrast to primary underflow and primary minimum-overlaps partitioning we find here that the CPU costs dominate on each one of the architectures. Therefore there is hardly any performance difference in that case.

Finally, we tried to compute performance marks for the five architectures in order to find the best one out. For that purpose, we normalised the performance results of table 10.15 in the following way: first, the costs with uniform lifespan partitioning (for a certain join and on a certain architecture) were assumed to represent a value of 100; then, the other cost values were transformed to express the costs in comparison to the 100 that represented the corresponding uniform lifespan partitioning value; finally the average per architecture was taken over all cost results. Table 10.19 shows the new numbers and figure 10.43 visualises the averages. The 4/4, 8/2 and 16/1 architectures are the clear winners in that comparison – a conclusion that has already been drawn from the results of section 8.5.

$\mathcal{M}$ / $\mathcal{N}$	$R \bowtie_C R$			$R \bowtie_C Q$			$Q \bowtie_C Q$		
	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps
1 / 16	1904	1939	1927	1858	1448	1560	2514	2540	2540
2 / 8	1623	1000	1215	1782	1196	1382	2435	1329	1523
4 / 4	1598	809	1074	1765	1155	1365	2426	974	1094
8 / 2	1585	803	1068	1746	1144	1355	2402	965	1085
16 / 1	1582	801	1066	1739	1139	1351	2390	961	1082

Table 10.15: The performance results (in sec.) for the three joins on varying parallel architectures.

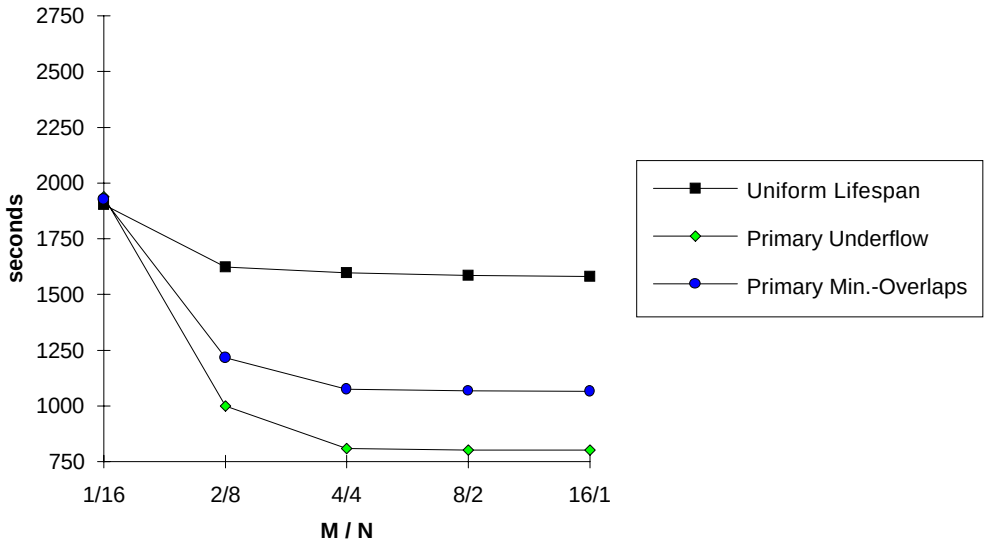


Figure 10.36: Performance results for the $R \bowtie_C R$ on varying parallel architectures.

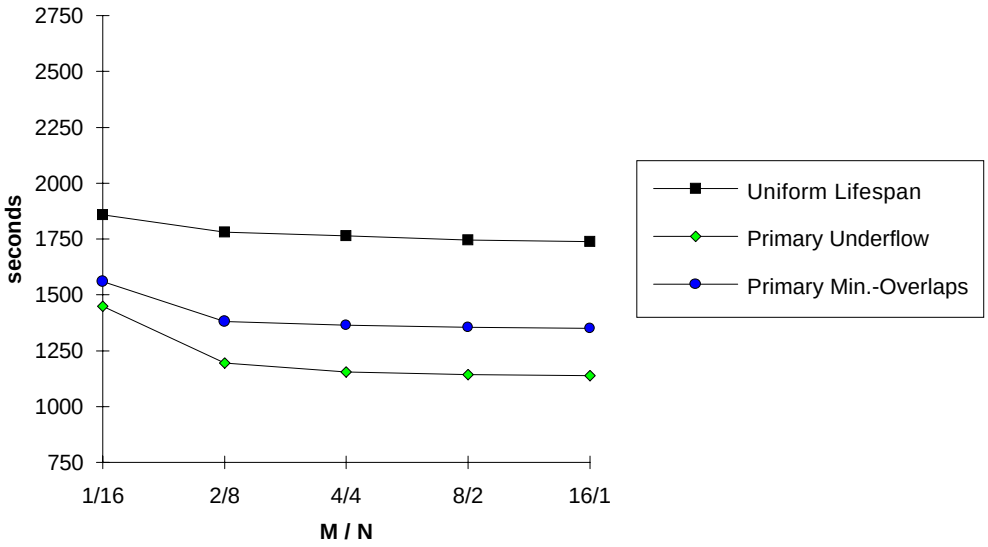


Figure 10.37: Performance results for the $R \bowtie_C Q$ on varying parallel architectures.

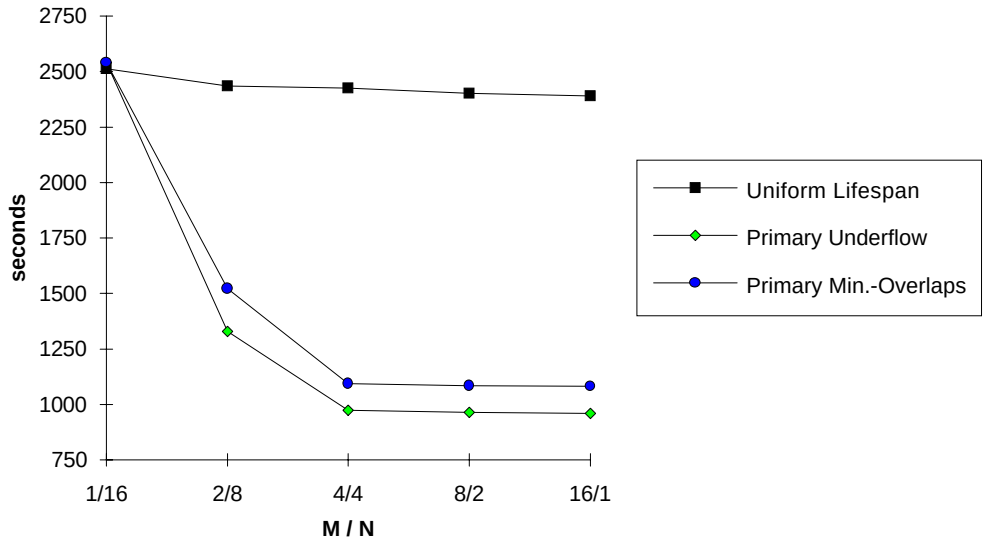


Figure 10.38: Performance results for the $Q \bowtie_C Q$ on varying parallel architectures.

	Parameters				$C_{2(a)}$			$C_{2(b)}$			$C_{2(c)}$				
	$\mathcal{M}$	$\mathcal{N}$	m	C_{part}	I/O	CPU	Mem	I/O	CPU	Mem	I/O	CPU	Mem	C_{join}	C_{total}
Uniform Lifespan	1	16	384	86	36	729	845	0.0	113	128	36	729	845	1818	1904
	2	8	384	52	23	729	573	0.0	113	91	23	729	573	1571	1623
	4	4	384	27	12	729	329	0.8	113	51	12	729	329	1571	1598
	8	2	384	14	6	729	176	0.5	113	27	6	729	176	1571	1585
	16	1	384	11	5	729	174	0.5	113	27	5	729	174	1571	1582
Primary Underflow	1	16	124	51	21	378	873	2.9	39	143	21	378	873	1888	1939
	2	8	124	26	11	378	450	1.5	39	73	11	378	450	974	1000
	4	4	124	15	6	378	265	0.8	39	37	6	378	265	794	809
	8	2	124	9	4	378	154	0.4	39	19	4	378	154	794	803
	16	1	124	7	2	378	90	0.2	39	9	2	378	90	794	801
Primary Min.-Overlaps	1	16	124	51	20	442	818	2.9	175	240	20	442	818	1876	1927
	2	8	124	28	12	442	506	1.7	175	145	12	442	506	1187	1215
	4	4	124	15	6	442	255	1.0	175	91	6	442	255	1059	1074
	8	2	124	9	4	442	176	0.5	175	50	4	442	176	1059	1068
	16	1	124	7	2	442	106	0.4	175	42	2	442	106	1059	1066

Table 10.16: The performance component results (in sec.) for $R \bowtie_C R$ on varying parallel architectures.

	Parameters				$C_{2(a)}$			$C_{2(b)}$			$C_{2(c)}$				
	$\mathcal{M}$	$\mathcal{N}$	m	C_{part}	I/O	CPU	Mem	I/O	CPU	Mem	I/O	CPU	Mem	C_{join}	C_{total}
Uniform Lifespan	1	16	384	86	36	1057	600	0.0	73	52	36	596	643	1772	1858
	2	8	384	56	26	1057	449	0.0	73	31	21	596	399	1725	1782
	4	4	384	40	23	1057	441	1.6	73	30	13	596	390	1725	1765
	8	2	384	21	12	1057	259	0.9	73	18	7	596	204	1725	1746
	16	1	384	13	7	1057	252	0.5	73	17	5	596	142	1725	1739
Primary Underflow	1	16	196	67	29	670	601	2.9	47	60	27	414	651	1381	1448
	2	8	196	44	22	670	538	2.0	47	45	15	414	436	1152	1196
	4	4	196	24	12	670	275	1.0	47	24	7	414	232	1130	1155
	8	2	196	14	7	670	250	0.6	47	19	5	414	156	1130	1144
	16	1	196	9	4	670	160	0.3	47	11	3	414	99	1130	1139
Primary Min.-Overlaps	1	16	183	63	27	779	599	2.9	73	71	24	490	645	1497	1560
	2	8	183	40	18	779	381	1.8	73	36	14	490	364	1342	1382
	4	4	183	23	12	779	308	1.1	73	33	7	490	276	1342	1365
	8	2	183	13	6	779	220	0.6	73	19	5	490	172	1342	1355
	16	1	183	9	3	779	186	0.4	73	18	2	490	117	1342	1351

Table 10.17: The performance component results (in sec.) for $R \bowtie_C Q$ on varying parallel architectures.

	Parameters				$C_{2(a)}$			$C_{2(b)}$			$C_{2(c)}$				
	$\mathcal{M}$	$\mathcal{N}$	m	C_{part}	I/O	CPU	Mem	I/O	CPU	Mem	I/O	CPU	Mem	C_{join}	C_{total}
Uniform Lifespan	1	16	384	86	36	1141	1164	0.0	92	100	36	1141	1164	2428	2514
	2	8	384	61	27	1141	838	1.9	92	61	27	1141	838	2374	2435
	4	4	384	52	24	1141	817	1.6	92	59	24	1141	817	2374	2426
	8	2	384	28	13	1141	458	0.9	92	37	13	1141	458	2374	2402
	16	1	384	16	7	1141	272	0.5	92	22	7	1141	272	2374	2390
Primary Underflow	1	16	124	64	27	457	1166	2.9	40	143	27	457	1166	2476	2540
	2	8	124	33	14	457	611	1.5	40	74	14	457	611	1296	1329
	4	4	124	21	9	457	419	0.8	40	37	9	457	419	953	974
	8	2	124	12	5	457	213	0.4	40	19	5	457	213	953	965
	16	1	124	9	2	457	109	0.2	40	10	2	457	109	953	961
Primary Min.-Overlaps	1	16	124	64	27	494	1161	2.9	86	153	27	494	1161	2476	2540
	2	8	124	34	14	494	691	1.8	86	106	14	494	691	1489	1523
	4	4	124	21	9	494	401	1.0	86	65	9	494	401	1073	1094
	8	2	124	12	5	494	221	0.5	86	40	5	494	221	1073	1085
	16	1	124	9	2	494	118	0.3	86	21	2	494	118	1073	1082

Table 10.18: The performance component results (in sec.) for $Q \bowtie_C Q$ on varying parallel architectures.

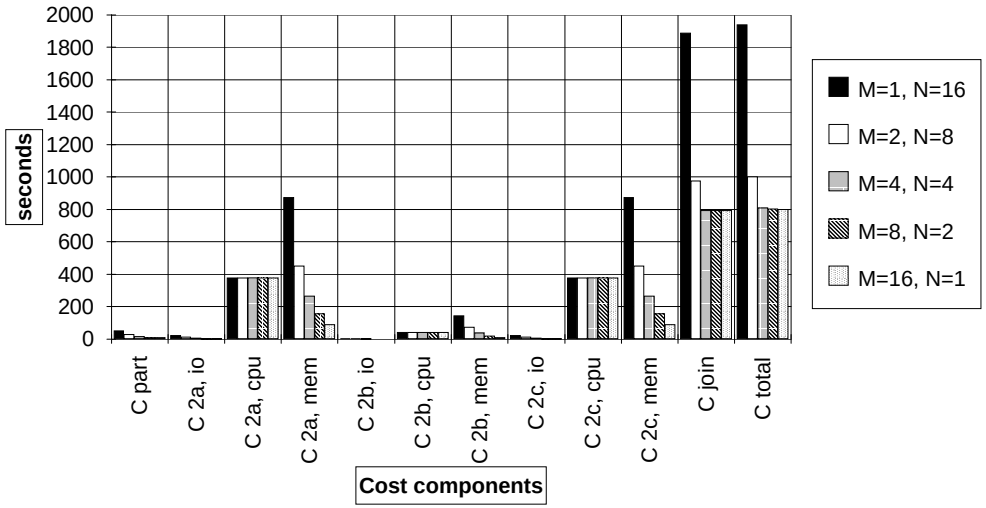


Figure 10.39: Cost components for $R \bowtie_C R$ using primary underflow partitioning.

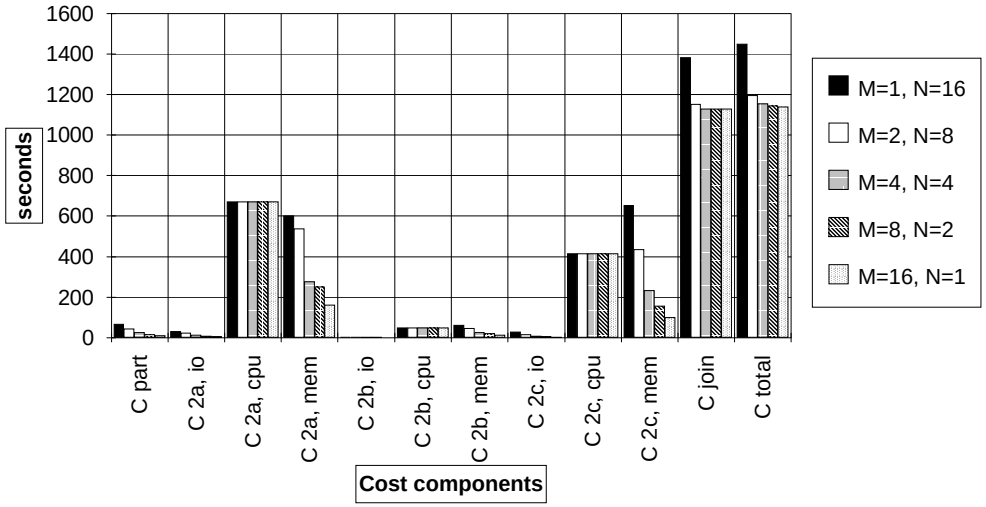


Figure 10.40: Cost components for $R \bowtie_C Q$ using primary underflow partitioning.

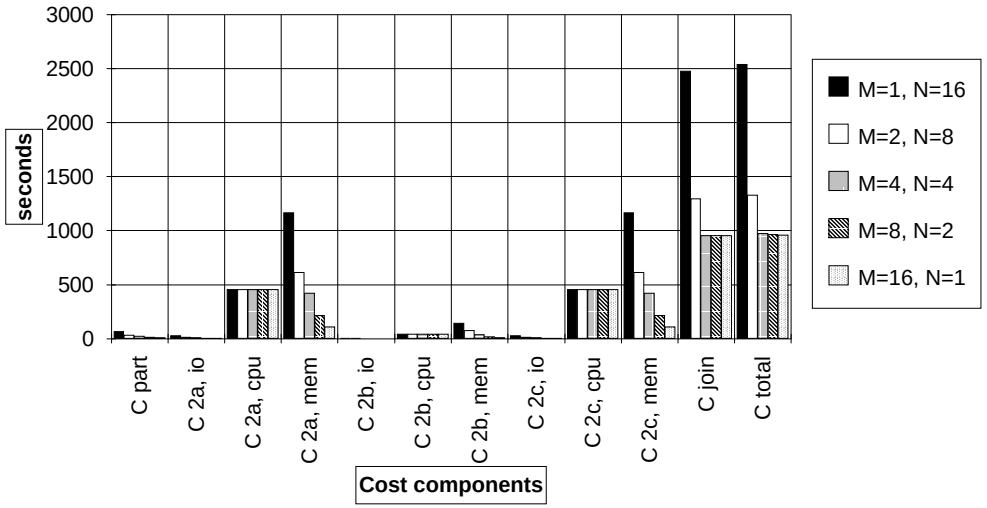


Figure 10.41: Cost components for $Q \bowtie_C Q$ using primary underflow partitioning.

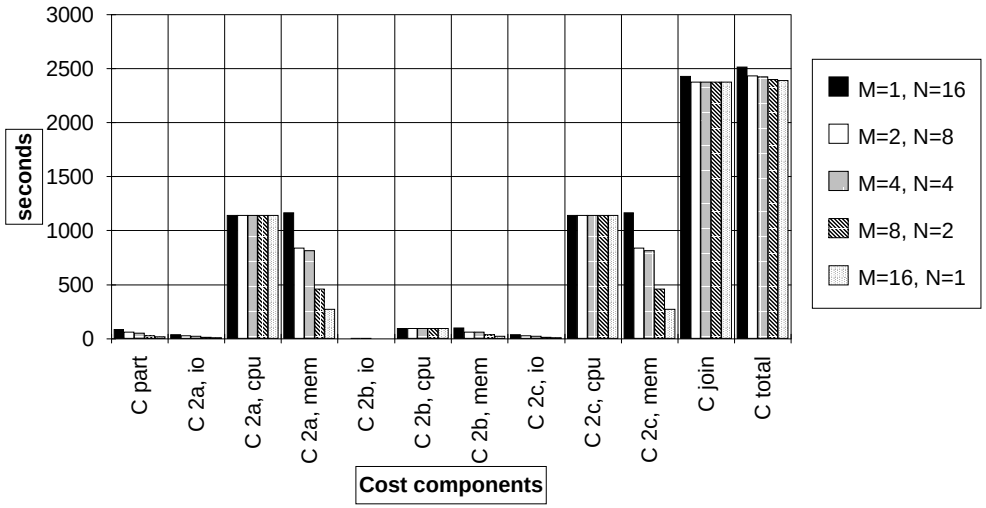


Figure 10.42: Cost components for $Q \bowtie_C Q$ using uniform lifespan partitioning.

$\mathcal{M}$ / $\mathcal{N}$	$R \bowtie_C R$			$R \bowtie_C Q$			$Q \bowtie_C Q$			average
	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps	uniform lifespan	primary underflow	primary min.-overlaps	
1/16	100	102	101	100	78	84	100	101	101	96
2/8	100	62	75	100	67	78	100	55	63	78
4/4	100	51	67	100	65	77	100	40	45	72
8/2	100	51	67	100	66	78	100	40	45	72
16/1	100	51	67	100	66	78	100	40	45	72

Table 10.19: The normalised performance results for the three joins on varying parallel architectures.

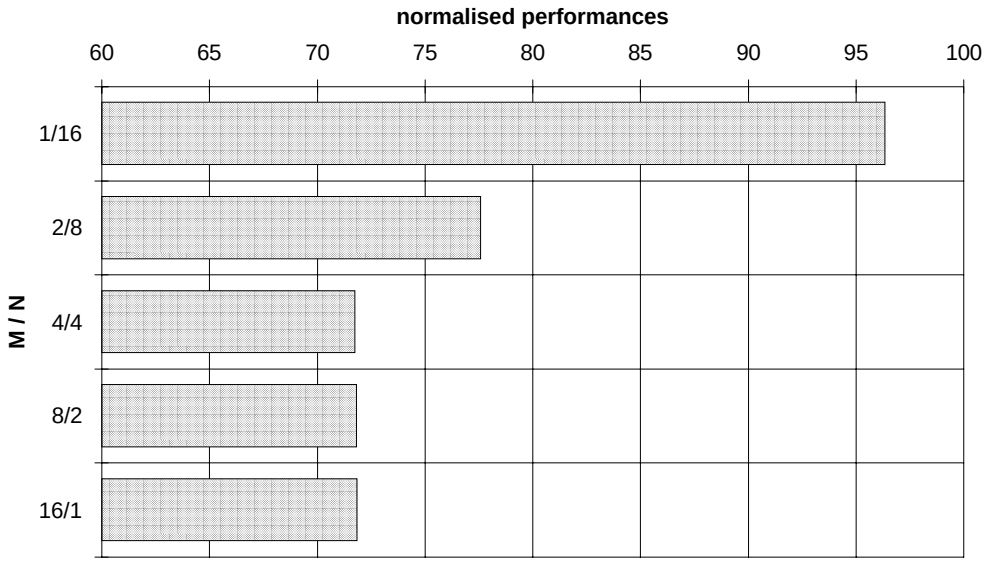


Figure 10.43: Comparison of the five parallel architectures.

10.8 Influence of the Condensation Factor a

In section 7.3.3 we proposed to 'condense' IP-tables in order to reduce their size. The idea was to collapse a of the original entries into one new entry. The new IP-table would have only $\frac{1}{a}$ of the entries and therefore only $\frac{1}{a}$ of the size of the original IP-table. A special form of condensation is the endpoint IP-table in which only those entries appear which correspond to interval endpoints. Theoretically, condensation can be expected to have two effects:

- The IP-table entries become coarser because they contain summarised information. For example, an entry does not contain the number of intervals starting at time t_j, i.e. $s_R(t_j)$ in a complete IP-table $I(R)$, but the number of intervals starting *between time* t_{j-1} *and time* t_j, i.e. $s'_R(t_j)$ in a condensed IP-table $I'(R, a)$. However, the underflow strategies use the $s_R(t)$ ($s'_R(t)$ respectively) values to approach the maximum number of tuples that are allowed per fragment[3]. The more condensed the IP-tables are the bigger are the steps with which an underflow strategy approaches the respective limit. This means that it might not come as close to the limit if it uses 'big' steps. This would be wasteful of effort expended in choosing carefully a near-optimal limit. Also, the difference between the loads of the fragments, i.e. the load imbalance, can be bigger for the same reason. Therefore, we can expect the underflow strategy to perform worse with an increasing value of a, especially on a parallel architecture where a good balance between the loads of the partial joins is essential in order to perform well.

  The effect on the minimum-overlaps strategies is difficult to predict. As mentioned before, their prime goal is to reduce the total number of overlaps. During this process, they possibly make concessions that worsen the load balance. It will be interesting to see whether the minimum-overlaps strategies perform better or worse with an increasing value for a.

- A clear advantage of condensation is that it reduces the sizes of the IP-tables. This has a direct impact on the time that is spent on the optimisation itself. The underflow strategies have a time complexity of $O(N)$ where N is the number of entries in the (joint) IP-table. The minimum-overlaps strategies have $O(N^2)$ while the time complexity of the uniform

[3] See *if*-statement in figures 9.7 (page 224) and 9.9 (page 226).

partitioning strategies does not depend on the IP-tables' sizes. We can therefore expect the underflow and minimum-overlaps strategies to benefit from condensed IP-tables as they require less time to decide on a suitable partition.

We set up experiments that test the impact of condensation on the primary underflow and the minimum-overlaps strategies (both with $X_R = 0.0075 \cdot |R|$, $X_Q = 0.0075 \cdot |Q|$) when applied to the three joins. As usual, the experiments were conducted for a parallel architecture and on a single-processor machine. We used the set $\{\textit{end}, 1, 2, \ldots, 30, 32, 36, 40, 44, \ldots, 60\}$ as values for a; '*end*' is thereby used as a symbolic value meaning that the respective IP-table was condensed by using endpoints only. The performance results are listed in tables 10.20 and 10.21 and visualised in figures 10.44, 10.45, 10.46 and 10.47. Figures 10.44 and 10.45 contain a lot of noise which makes it difficult to observe a certain trend. We therefore converted them to the graphs of figures 10.48 and 10.49 in which a value for a certain join and a certain a was computed by taking the average of the four preceding a values and a itself ('moving average'). This process smoothes the graphs and makes the general trends more visible. Finally, there is figure 10.50 that shows the times that were spent on the optimisation process only. Times are only given for $a \leq 30$ as the times were almost constant for $a > 30$.

First, we analyse the graph of figure 10.48 which shows the performances for primary underflow partitioning on a parallel architecture. As expected, the performances become worse for an increasing a, but only for the joins $R \bowtie_C R$ and $Q \bowtie_C Q$. For the join $R \bowtie_C Q$ there is a rather unexpected effect: for most values of a, the performance is slightly better in comparison to $a = 1$ or $a = \textit{end}$. This is rather surprising and cannot be explained from an algorithmic point of view. Its origin lies either in the data itself or in the fact that the chosen X_R and X_Q values (see above) were not optimal for $R \bowtie_C Q$ and that the summarising effect of condensation corrected this a little bit as outlined above. Also, when processing $R \bowtie_C R$, $R \bowtie_C Q$ and $Q \bowtie_C Q$, the strategies use the timepoint sets

$$\begin{aligned} V'(R,a) \cup V'(R,a) &= V'(R,a) \\ V'(R,a) \cup V'(Q,a) & \\ V'(Q,a) \cup V'(Q,a) &= V'(Q,a) \end{aligned}$$

respectively. However, the set $V'(R,a) \cup V'(Q,a)$ obviously contains much more timepoints to choose from than $V'(R,a)$ or $V'(Q,a)$. Because of condens-

ation there are hardly coinciding timepoints which leads to

$$|V'(R,a) \cup V'(Q,a)| \;\approx\; |V'(R,a)| + |V'(Q,a)|$$

for $a \geq 2$. This advantage translates into a much better resistance against the negative effects of condensation for the join $R \bowtie_C Q$ in comparison to the joins $R \bowtie_C R$ and $Q \bowtie_C Q$. Actually, we can observe this advantage in all the other charts in figures 10.49, 10.46 and 10.47 too.

Now we turn to figure 10.49 which shows the averaged performances on the parallel architecture for the primary minimum-overlaps strategy depending on a. We note that there is a rather positive effect of condensation in the first part of the chart. The best performances are achieved around $a = 16$ (for $R \bowtie_C R$), $a = 22$ (for $R \bowtie_C Q$) and $a = 19$ (for $Q \bowtie_C Q$). Previously, we observed that there is a relatively severe penalty caused by the concessions that the strategy makes with respect to the load balance in order to achieve a minimum number of overlaps. It seems that the strategy can make less concessions (a) if there are less timepoints from which it can choose and (b) if the values $s_R(t)$ and $s_Q(t)$[4] are larger. Naturally, if the choice becomes too restricted then there is a negative impact on the performance. Therefore, there is a tradeoff between restricting the choice a little bit but not too much. This is exactly what can be seen in the graphs of figure 10.49.

The scenery changes when it comes to the single-processor architecture (figures 10.46 and 10.47). For the joins $R \bowtie_C R$ (join 1) and $Q \bowtie_C Q$ (join 3), the impact of an increasing condensation of the IP-tables on the performance is generally negative[5]. Again, processing of the join $R \bowtie_C Q$ (join 2) resists much better against the negative effects of condensation because of the reasons mentioned above. We can even observe a slight performance benefit which is probably caused by the same reasons as mentioned earlier.

Finally, we turn our attention to the effect that condensed IP-tables have on the performance of the optimisation itself. Figure 10.50 shows the times that the optimisation process took when being run on a two-processor Sun SS20 computing server. As expected, we see that the primary minimum-overlaps partitioning benefits most of it due to its time complexity of $O(N^2)$. It is interesting that – in terms of elapsed time – it is roughly as fast as primary underflow partitioning for $a \geq 10$.

[4] ... or $s'_R(t)$ and $s'_Q(t)$ in the case of condensed IP-tables.

[5] with one single exception, i.e. for join 1, $a = 16$ and the primary minimum-overlaps strategy (figure 10.47).

In summary, it is fair to say that experiments show that condensation does not only have positive effects on the performance the optimisation process itself but also that the anticipated negative effects are not that severe at all. The primary minimum-overlaps strategy in particular can draw performance benefits when using condensed IP-tables. Generally, it is possible to say that a condensation factor a between approx. 10 and 20 can improve the performances in many cases or at least does not severely penalise the performances in the other cases. This is an interesting result with respect to section 7.3 in which we addressed the problem of IP-table size. It means that the sizes of IP-tables which were already comparable or lower than those of data samples could be further decreased to between $\frac{1}{10}$ to $\frac{1}{20}$ of the original size without paying a severe performance penalty and sometimes even improving the join performances.

a	Primary Underflow			Primary Min.-Overlaps		
	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)
end	716	1141	896	1111	1336	1114
1	716	1141	896	1111	1336	1114
2	807	1131	976	1085	1327	1109
3	795	1178	962	1026	1346	1063
4	801	1140	957	1007	1302	1044
5	777	1105	947	979	1234	1014
6	758	1144	921	968	1237	948
7	819	1083	925	981	1229	1021
8	780	1107	889	932	1299	1063
9	787	1138	1026	920	1302	1028
10	823	1063	950	935	1139	926
11	825	1091	942	986	1282	975
12	791	1096	1004	931	1318	1045
13	814	1095	1020	908	1205	1117
14	749	1089	1067	848	1180	1058
15	802	1160	1041	853	1217	966
16	771	1038	1006	936	1176	970
17	793	1050	913	952	1207	1039
18	792	1179	933	964	1122	960
19	820	1117	966	985	1132	1019
20	788	1242	1002	959	1128	1055
21	755	1090	1041	1049	1159	1101
22	804	1055	1027	1024	1111	1144
23	774	1016	1061	1011	1301	1100
24	864	1036	1047	1079	1274	1141
25	859	999	1027	1065	1172	1166
26	762	1030	1089	1076	1188	1112
27	856	1061	1098	1097	1175	1154
28	918	1102	1179	1117	1147	1173
29	1000	1009	1092	1087	1206	1096
30	959	1073	1114	1054	1067	1152
32	871	1103	1236	969	1069	1227
36	1015	1057	1249	1134	1071	1238
40	971	1064	1217	1161	1196	1196
44	1070	1016	1327	1331	1283	1283
48	1058	1116	1260	1334	1154	1265
52	1139	1043	1352	1096	1277	1299
56	1140	1165	1420	1410	1201	1420
60	1237	1235	1288	1643	1212	1350

Table 10.20: Performance results (in sec.) on the parallel architecture depending on a varying condensation factor a for the IP-tables.

a	Primary Underflow			Primary Min.-Overlaps		
	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)	$R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)
a	Join 1	Join 2	Join 3	Join 1	Join 2	Join 3
end	7898	5560	10410	7877	5362	10410
1	7898	5560	10410	7877	5361	10410
2	7897	5555	10406	7880	5572	10406
3	7934	5550	10402	7882	5564	10400
4	7922	5554	10394	7888	5567	10394
5	7976	5550	10394	7888	5564	10389
6	7970	5541	10390	7900	5371	10383
7	7961	5531	10381	7898	5371	10379
8	7985	5532	10378	7894	5562	10366
9	7993	5377	10370	7896	5373	10379
10	7957	5386	10366	7898	5370	10361
11	7991	5378	10365	7914	5546	10345
12	8005	5377	10372	7913	5380	10355
13	8013	5363	10368	7916	5376	10365
14	8009	5381	10354	7922	5389	10359
15	8037	5380	10344	7915	5388	10341
16	8034	5386	10331	7556	5390	10320
17	8065	5388	10331	7922	5389	10311
18	8088	5396	10332	7928	5402	10311
19	8098	5382	10333	7952	5389	10314
20	8107	5360	10330	7977	5371	10311
21	8144	5377	10332	7999	5400	10313
22	8177	5367	10344	8023	5397	10322
23	8191	5368	10353	8047	5404	10331
24	8204	5377	10363	8075	5419	10342
25	8214	5368	10369	8087	5414	10353
26	8245	5367	10382	8115	5415	10364
27	8282	5374	10393	8140	5421	10377
28	8314	5354	10405	8163	5404	10392
29	8321	5350	10416	8188	5376	10403
30	8335	5354	10429	8215	5404	10418
32	8395	5358	10457	8259	5390	10441
36	8490	5362	10511	8370	5380	10499
40	8583	5335	10567	8468	5344	10554
44	8687	5306	10627	8574	5313	10617
48	8785	5321	10685	8669	5316	10677
52	8859	5306	10744	8770	5302	10735
56	8961	5325	10807	8863	5314	10796
60	9044	5293	10871	8952	5281	10863

Table 10.21: Performance results (in sec.) on the single-processor machine depending on a varying condensation factor a for the IP-tables.

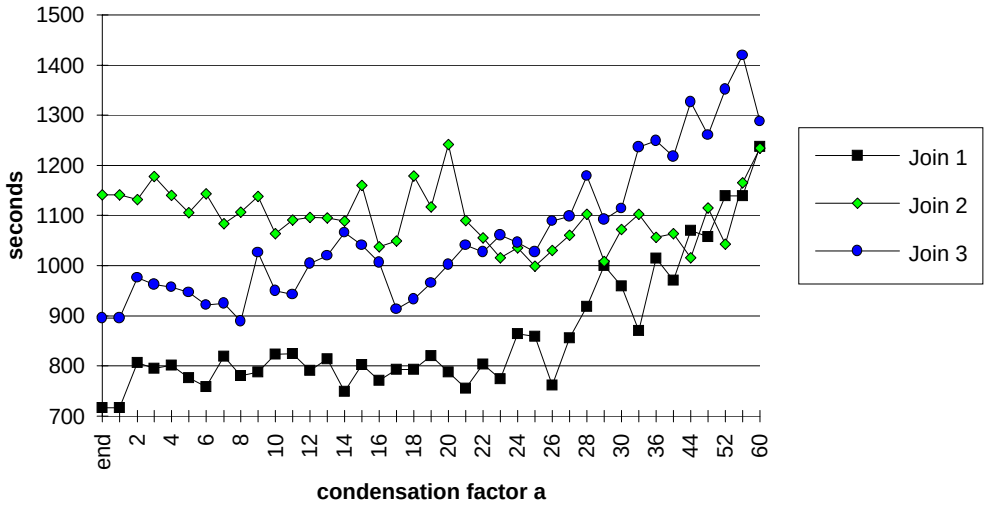


Figure 10.44: Performances for primary underflow partitioning on the parallel architecture and depending on a.

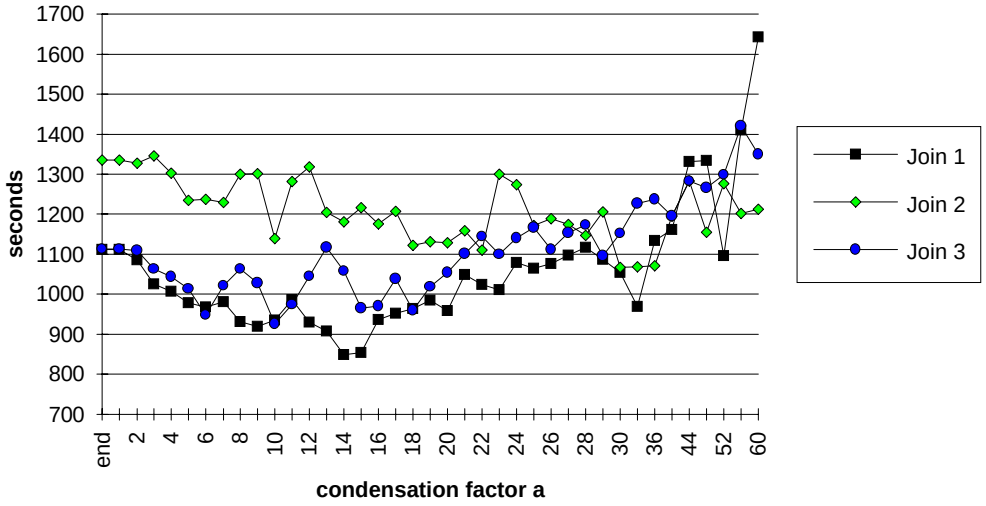


Figure 10.45: Performances for primary min.-overlaps partitioning on the parallel architecture and depending on a.

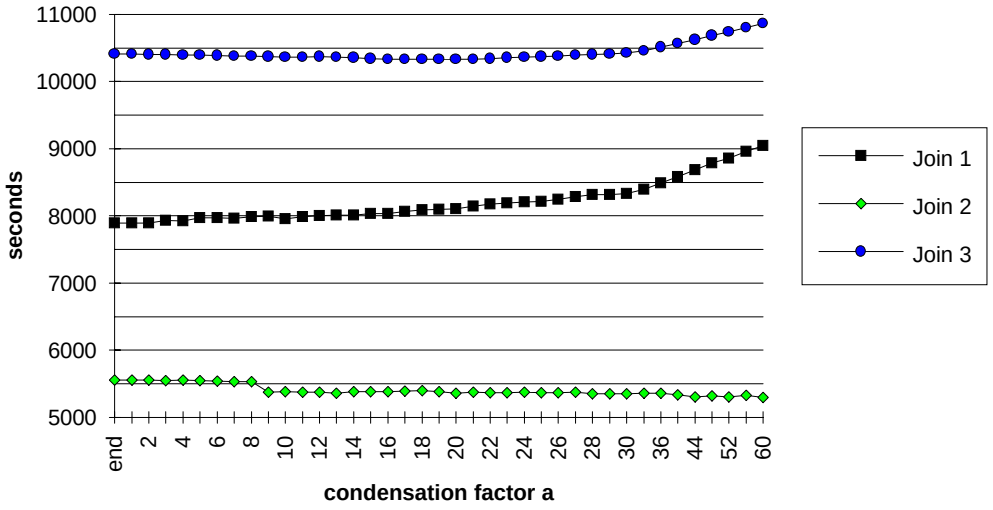


Figure 10.46: Performances for primary underflow partitioning on a single-processor machine and depending on a.

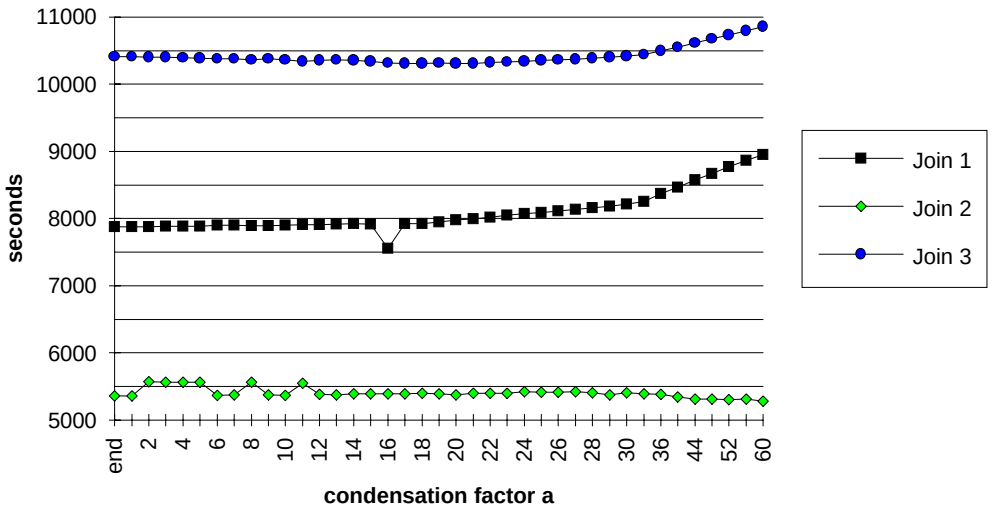


Figure 10.47: Performances for primary min.-overlaps partitioning on a single-processor machine and depending on a.

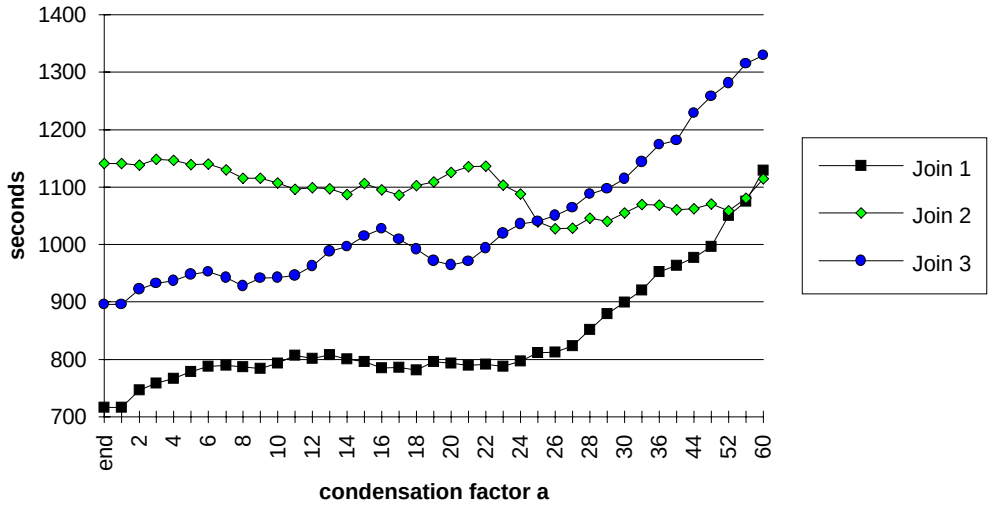


Figure 10.48: Performance, expressed as moving averages for primary underflow partitioning on the parallel architecture, varying on a.

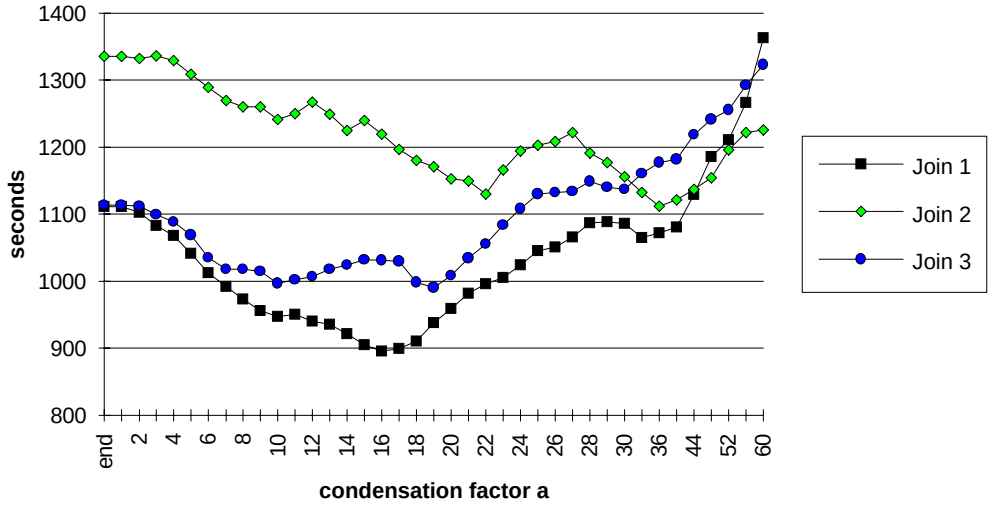


Figure 10.49: Performance, expressed as moving averages for primary min.-overlaps partitioning on the parallel architecture, varying on a.

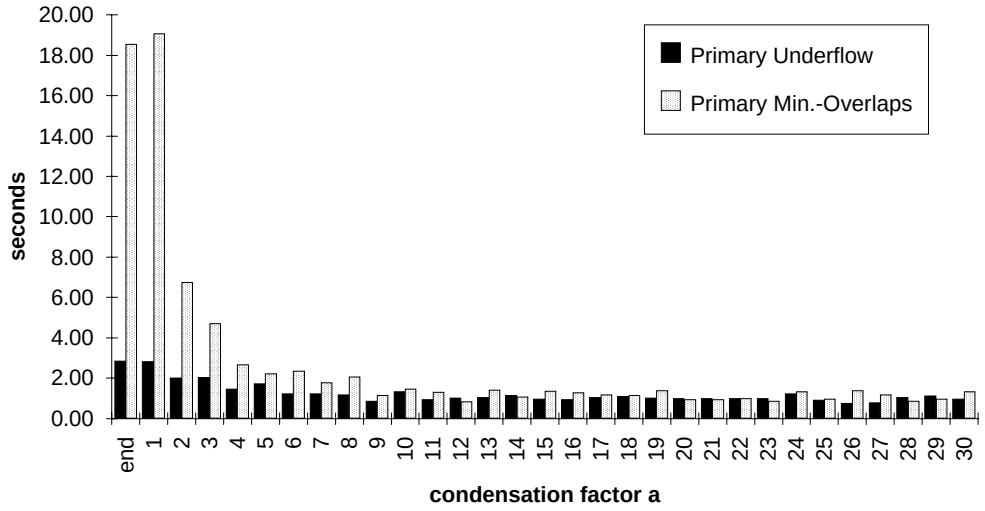


Figure 10.50: Times for the optimisation process on a Sun SS20 for a varying a.

10.9 Impact of Black-Out Preprocessing

In section 9.4, we presented the *black-out preprocessing strategy* that essentially scraps all entries of an IP-table $I(R)$ whose associated $o_R(t)$ value lies beyond a certain threshold value Y. These entries are considered as unsuitable to provide a good breakpoint for a partition. Therefore, an underflow strategy cannot choose such unsuitable breakpoints if the IP-tables that it uses have been preprocessed in this way.

The experiments of section 10.2 showed discouraging results for black-out preprocessing. However, this might be due to a bad choice for the threshold Y which was chosen to be the average $\bar{O}$ of all $o_R(t)$ values in the respective IP-table $I(R)$ according to equation (10.1). Here, we want to look at black-out preprocessing in more detail. Experiments were set up in which the threshold Y was varied. For practical purposes we express Y in percent of $\bar{O}$. A value of 70%, for example, is supposed to mean that $Y = 0.7 \cdot \bar{O}$. The respective Y' threshold was set to be 10% below the corresponding X_R and X_Q values, i.e.

$$\begin{aligned} Y' &= 0.9 \cdot X_R = 0.9 \cdot 0.0075 \cdot |R| \\ &= 0.9 \cdot X_Q = 0.9 \cdot 0.0075 \cdot |Q| \\ &= 0.9 \cdot 0.0075 \cdot 121728 = 821.7 \end{aligned}$$

Table 10.22 shows the performance results on the parallel and on the single-processor architectures. Figures 10.51 and 10.52 show the differences with respect to the time that is required if black-out preprocessing is not used in the partitioning process.

Not surprisingly, we find that the lower the threshold Y is, the higher is the impact of black-preprocessing on the performances, at least on the parallel architecture. The performance gains, if there are any, remain marginal with 5% at best. From figures 10.51 and 10.52 we can see that black-out preprocessing is often beneficial for the performances for the 'mixed' join $R \bowtie_C Q$ while we find a negative effect on the performances for the 'self' joins $R \bowtie_C R$ and $Q \bowtie_C Q$. The latter effect is due to two slightly different reasons:

- The periodic profile of R, especially with many sharply rising flanks, already provides many opportunities for choosing 'good' breakpoints, i.e. ones with a low $o_R(t)$ value. Therefore, most X_R and X_Q values cause the ordinary primary underflow strategy (i.e. without black-out preprocessing) to produce breakpoints that lie on or close to the bottoms of the valleys. This was, for example, the case in our experiments. But

this means that ordinary primary underflow partitioning already produces a good partition. The introduction of black-out preprocessing can then only have a negative effect by restricting the primary underflow strategies' choice on breakpoints and leads to a bad load balance. This also explains that there is hardly any negative impact on the performances for $R \bowtie_C R$ on the single-processor machine because there, the load balance is by far not as important.

- The non-periodic profile of Q forces the ordinary primary underflow strategy to choose breakpoints with high $o_Q(t)$ values. Black-out preprocessing can change this but possibly on the expense of worsening the load balance between the fragments. However, we have already seen that the load balance is the most important factor for the performances on the parallel architecture. Again, the negative impact is reduced on the single-processor machine which underlines the significance of the load balance factor.

In contrast to the joins that have been discussed above, we note that the performances for the join $R \bowtie_C Q$ are positively affected by black-out preprocessing, although one has to stress that the improvement is modest. It is due to the reduction of overlapping intervals – the initial idea of black-out preprocessing – without affecting the load balance negatively.

In summary, we can conclude that the positive influence of black-out preprocessing are lower than we had hoped for when designing this strategy in section 9.4. In a considerable amount of cases it restricts the primary underflow strategy too much in its choice and causes a load balance which is worse than in the original case. This is usually penalised when a join is processed on a parallel architecture. In contrast, there is a modest benefit in most cases if the join is processed sequentially.

Y (in % of $\bar{O}$)	parallel architecture $R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)	single-processor machine $R \bowtie_C R$ (Join 1)	$R \bowtie_C Q$ (Join 2)	$Q \bowtie_C Q$ (Join 3)
no black-out	716	1141	896	7898	5560	10410
120%	716	1150	896	7898	5562	10410
110%	716	1133	896	7898	5384	10410
100%	712	1133	898	7892	5385	10411
90%	723	1140	895	7889	5564	10412
80%	737	1132	901	7917	5384	10414
70%	754	1132	904	7926	5376	10413
60%	750	1142	910	7904	5560	10416
50%	752	1092	911	7908	5571	10420
40%	764	1087	911	7909	5578	10443
30%	781	1091	937	7936	5594	10475

Table 10.22: Performance results (in sec.) depending on Y for the three joins and the primary underflow strategy using black-out preprocessing.

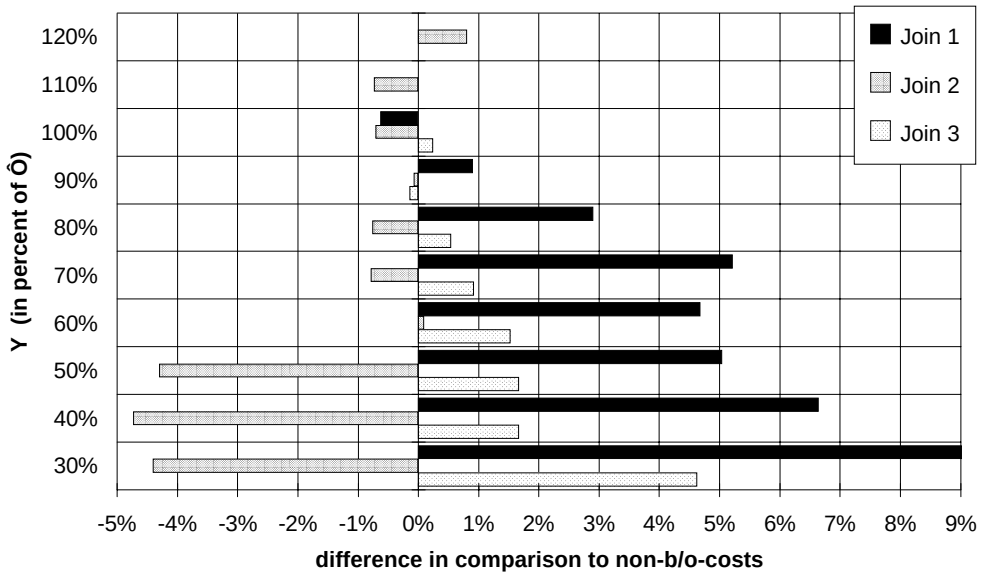


Figure 10.51: Performance differences for primary underflow partitioning with black-out preprocessing on the parallel architecture.

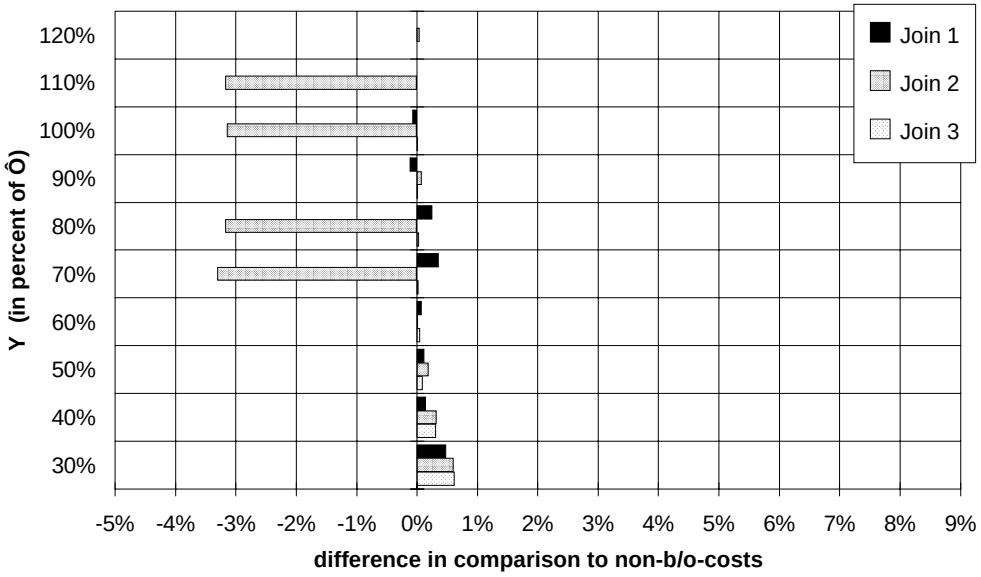


Figure 10.52: Performance differences for primary underflow partitioning with black-out preprocessing on the single-processor machine.

10.10 Summary

We will now summarise the principal conclusions that can be drawn from the experiments of sections 10.2 to 10.9.

10.10.1 Experiments on the Parallel Architecture

On the parallel architecture with $\mathcal{M} = 4$ and $\mathcal{N} = 4$, primary underflow partitioning produced the best performances in almost every situation. Exceptions were high values of τ (see figures 10.24 and 10.25) where primary minimum-overlaps partitioning performed better. This does not mean that the primary underflow strategy is the strategy to choose in every case but it does allow to conclude that well balanced partial join computations – this is the goal of the primary underflow strategy – are much more important for the join performance than a reduced number of overlapping intervals – the goal for which the primary minimum-overlaps strategy aims. The uniform partitioning strategies could not match the performances of the primary underflow and minimum-overlaps strategies. The costs were at times twice as high in comparison to primary underflow and minimum-overlaps partitioning (see tables 10.4 and 10.6, for example). This underlines our initial assumption that a naive partitioning approaches end in poor performances and that more sophisticated strategies, such as the underflow and minimum-overlaps strategies, are required.

The experiments of section 10.4 showed that, for a temporal intersection join $R \bowtie_C Q$, X_R and X_Q values with

$$\begin{aligned} X_R &\approx \frac{Z}{100} \cdot |R| \\ X_Q &\approx \frac{Z}{100} \cdot |Q| \end{aligned}$$

generally deliver the best or at least near-best performance results for $Z \leq 1$.

A further, very significant conclusion is that condensation is not as harmful as we previously expected. In some cases it even improved the performances, especially for the primary minimum-overlaps strategies (see figure 10.49). A condensation factor a between 10 and 20 is feasible. We remember that this means that IP-tables can be reduced to between $\frac{1}{10}$-th to $\frac{1}{20}$-th of their original sizes. This not only accelerates the optimisation process (see figure 10.50) but also has many positive effects with respect to storage and maintenance of IP-tables.

The impact of black-out preprocessing largely depends on the profiles of the relations that participate in the join. On the parallel architecture, performances

could improve up to 4% but also decrease up to 9% depending on the actual situation. The experiments in section 10.9 are certainly not sufficient to provide clear guidelines when and when not to use black-out preprocessing. However, the 4% increase and 9% decrease in performance suggest that the margins for improvements are only minor.

10.10.2 Experiments on the Single-Processor Architecture

On the single-processor architecture, uniform lifespan partitioning surprisingly produces the best performance results in many situations as long as the value for m is high enough (i.e. approx. $m \geq 500$). For primary underflow and minimum-overlaps partitioning the best choice of X_R and X_Q values for a join $R \bowtie_C Q$ is given by the rule

$$\begin{aligned} X_R &\approx \frac{Z}{100} \cdot |R| \\ X_Q &\approx \frac{Z}{100} \cdot |Q| \end{aligned}$$

for $Z \leq 1$, according to the experiments of section 10.4.

Condensation is similarly successful on the single-processor machine as on the parallel architecture. Primary underflow as well as primary minimum-overlaps partitioning is hardly affected by condensation approx. for $a \leq 30$. In some cases we even observed some minor performance improvements (see figures 10.46 and 10.47). This is again very encouraging due to the many advantages that can be drawn from smaller IP-tables.

In contrast to the parallel case, the negative impact of black-out preprocessing on the join performances were only minor (around 0.6% performance decrease for the worst case). In some situations, however, improvements of up to 3.3% were observed. This is again relatively modest.

Chapter 11

Using IP-Tables for Selectivity Estimation

11.1 Introduction

So far, the main purpose of IP-tables has been the efficient support for the optimisation of partitioned temporal join processing. In this chapter, we want to show that the scope and the applicability of IP-tables goes beyond that. In fact, IP-tables can be considered as a general metadata-structure that can be used in a wide range of temporal query optimisation techniques, especially in those that require (semi-) optimal partitioning of temporal data over a timestamp attribute, such as physical data partitioning or balancing temporal index structures.

In this chapter, we concentrate on an optimisation issue which is not directly connected to temporal data partitioning but that can nevertheless be efficiently supported by IP-tables, namely the selectivity estimation of temporal conditions. Selectivity estimation is a powerful way to predict the result sizes for many operations. On the basis of these predictions, an optimiser can then take many performance-relevant decisions, such as

- Which algorithm should be used to perform the respective operation most efficiently? Often there is a wide range of algorithms available, with some being suitable for low selectivities, others for high selectivities. The join operation is a good example for this: in chapter 3 we saw a variety of join algorithms with the nested-loops approach being suitable only for high selectivities whereas sort-merge or hash algorithms were more efficient in the case of low selectivities.

- Which is an efficient order in which operations should be processed? A query that incorporates various operations is often translated into an op-

erator tree. In this tree, operations with low selectivities (i.e. small result sizes) are moved near the leaves (i.e. these operations are performed first) and operations with high selectivities towards the root. This has the advantage that the initially-processed operations already discard large amounts of data and therefore reduce the sizes of the intermediate results. This can reduce the overall performance by a considerable amount.

- How many resources are required to process the query? From a system's point of view, the optimiser might want to consider the impact of the query on the system's overall performance: if the query is 'heavy' (i.e. very resource-consuming), for example, then other, 'lighter' queries might be granted priority.

Furthermore, an optimiser could warn a user if a query result size will be huge, and therefore possibly useless or not what has been intended; the user can then think of rewriting his / her query without a useless query being processed by the system. This is particularly relevant in the context of data mining or decision support systems which are likely to issue complex, ad-hoc queries involving huge amounts of data.

We restrict ourselves to the discussion of selectivities of temporal join conditions as these have been the focus of major parts of this thesis. This restriction is also sufficient to prove the wide applicability of IP-tables. As defined in section 3.4, the *selectivity factor* (or *selectivity*, for short) of a join $R \bowtie_C Q$ is

$$\text{(join) selectivity} = \frac{\text{size of the join result}}{\text{size of the cartesian product}} = \frac{|R \bowtie_C Q|}{|R| \cdot |Q|} \qquad (11.1)$$

In section 11.2, we classify temporal join conditions. Actually this is a summary of what was discussed in section 4.1 but this time with a slightly different emphasis. In section 11.3, we derive equations that allow either the exact calculation, or a reasonably accurate estimation of the size of temporal join results and therefore also the selectivity of the corresponding temporal join condition.

11.2 Temporal Join Conditions

We now want to recapitulate the discussion on temporal join conditions in section 4.1 and focus on the way in which they can be decomposed. Previously, we classified temporal joins according to their corresponding temporal join conditions. Thus, when speaking about decomposing a certain class / type of

temporal join we often refer to the decomposition of the underlying join condition (and vice versa). There is a set of elementary (i.e. non-decomposable) temporal joins / join conditions which is presented in section 11.2.1. On top of these, several composite conditions can be created, in particular those that arise from Allen's interval relationships (see section 11.2.2). We note that the classification scheme used here differs slightly from the one in section 4.1 because we focus on whether a condition is elementary or composite.

11.2.1 Elementary Conditions

As previously mentioned, there are many possible relationships between two intervals: one interval can lie completely *before* the other, both intervals can *start* and / or *end* at the same time, they can *overlap* each other etc. Temporal joins can be classified according to the type of interval relationship that its join condition is based on. Table 11.1 shows a set of join conditions. We treat them as elementary for the three reasons spelled out in section 4.1.
Please note that the after join is redundant within this list as

$$R \overset{\text{aft}}{\bowtie} Q \;=\; Q \overset{\text{bef}}{\bowtie} R$$

Allen's interval relationships [Allen, 1983] are frequently considered to be elementary too. However, they lead to more complex expressions when translated into relationships between intervals' start- and endpoints. This makes it more difficult to decompose complex temporal join conditions into elementary ones. For that reason we opt for the set presented in table 11.1.

11.2.2 Composite Conditions

Several temporal references in natural language translate into more complex interval relationships than those listed in table 11.1. For example, "same time" frequently means that time intervals have to intersect, i.e. they have to share a certain range, "during" means that one interval has to be entirely included into the other one, "exactly at the same time" implies that the intervals must be the same. Such conditions can be implemented by composing several of the elementary ones. Table 11.2 gives a list of such conditions and also shows the way in which they have been composed of elementary ones. This information will be useful in section 11.3.2 when the calculation of join result sizes and join selectivities of these types of joins is composed of the results of the elementary ones.

Relationship	**Join Name & Symbol**	**Condition**	**Informal Description**
start	start join: $\overset{\text{sta}}{\bowtie}$	$r.t_s = q.t_s$	same timestamp startpoints
finish	finish join: $\overset{\text{fin}}{\bowtie}$	$r.t_e = q.t_e$	same timestamp endpoints
meet	meet join: $\overset{\text{mt}}{\bowtie}$	$r.t_e = q.t_s$	timestamp of r ends where timestamp of q starts, i.e. they meet.
before	before join: $\overset{\text{bef}}{\bowtie}$	$r.t_e < q.t_s$	timestamp of r comes before q's timestamp
after	after join: $\overset{\text{aft}}{\bowtie}$	$r.t_s > q.t_e$	timestamp of r comes after q's timestamp
left-overlap	left-overlap join: $\overset{\text{lo}}{\bowtie}$	$r.t_s > q.t_s \ \wedge \ r.t_s < q.t_e$	startpoint of r's timestamp lies within q's timestamp
right-overlap	right-overlap join: $\overset{\text{ro}}{\bowtie}$	$r.t_e > q.t_s \ \wedge \ r.t_e < q.t_e$	endpoint of r's timestamp lies within q's timestamp

Additional constraints are: $r.t_s \leq r.t_e \ \wedge \ q.t_s \leq q.t_e$

Table 11.1: Elementary temporal joins and respective conditions for joining tuples $r \in R$ with $q \in Q$.

Join Name	**Composition**	**Informal Description**
equal join	$R \overset{=}{\bowtie} Q = R \overset{sta}{\bowtie} Q \cap R \overset{fin}{\bowtie} Q$	same timestamps
overlap join	$R \overset{olp}{\bowtie} Q = R \overset{lo}{\bowtie} Q \cup R \overset{ro}{\bowtie} Q$	timestamps overlap but do not start or finish at the same point
contain join	$R \overset{con}{\bowtie} Q = (R \overset{lo}{\bowtie} Q \cap R \overset{ro}{\bowtie} Q) \cup (R \overset{sta}{\bowtie} Q \cap R \overset{ro}{\bowtie} Q) \cup (R \overset{lo}{\bowtie} Q \cap R \overset{fin}{\bowtie} Q)$	timestamp of an $r \in R$ contains the entire timestamp of a $q \in Q$
during join	$R \overset{dur}{\bowtie} Q = Q \overset{con}{\bowtie} R$	timestamp of an $r \in R$ is required to lie entirely within the timestamp of a $q \in Q$
intersection join	$R \overset{int}{\bowtie} Q = R \overset{lo}{\bowtie} Q \cup R \overset{ro}{\bowtie} Q \cup R \overset{sta}{\bowtie} Q \cup R \overset{fin}{\bowtie} Q \cup R \overset{mt}{\bowtie} Q \cup Q \overset{mt}{\bowtie} R$	timestamps intersect

Table 11.2: Examples of temporal join types that can be derived from the elementary ones.

11.3 Size and Selectivity Calculations

In this section, we want to show how selectivity factors and result sizes can be computed for temporal joins. Section 11.3.1 looks at the elementary temporal joins that arise from section 11.2.1. In section 11.3.2, these results are used for calculating the selectivities for the composite types of section 11.2.2.

11.3.1 Elementary Joins

We now show how the result size – and therefore also the selectivity factor according to (11.1) – of the elementary temporal joins of table 11.1 can be calculated. For notational purposes, we assume that the selectivity factor / result size of some join $R \bowtie_C Q$ is to be derived. The set $V(R \cup Q) = \{t_1, \ldots, t_N\}$ comprises the interval start- and endpoints of all the rows in both of the relations that participate in the join, i.e. R and Q.

In the case of a start join $R \overset{\text{sta}}{\bowtie} Q$ we are looking for combinations $r \circ q$ of tuples $r \in R$ and $q \in Q$ whose timestamps start at the same time. There are $s_R(t)$ tuples in R and $s_Q(t)$ tuples in Q that start at a timepoint t. As $s_R(t) = s_Q(t) = 0$ for all $t \notin \{t_1, \ldots, t_N\}$ we can concentrate on the t_j for $j = 1, \ldots, N$. Considering that any timestamp has exactly one startpoint, we know that there are no redundant counts, therefore the result size can be computed by summing up the numbers. Thus the result size of a start join is

$$|R \overset{\text{sta}}{\bowtie} Q| = \sum_{j=1}^{N} s_R(t_j) \cdot s_Q(t_j)$$

Similarly, one can compute the result sizes for finish and meet joins:

$$|R \overset{\text{fin}}{\bowtie} Q| = \sum_{j=1}^{N} e_R(t_j) \cdot e_Q(t_j)$$

$$|R \overset{\text{mt}}{\bowtie} Q| = \sum_{j=1}^{N} e_R(t_j) \cdot s_Q(t_j)$$

A before join $R \overset{\text{bef}}{\bowtie} Q$ requires a timestamp of a tuple r to end before the timestamp of q starts if they are to be combined and put into the join result. Thus those tuples in R that end at timepoint t combine with all those tuples in Q that start after t, i.e.

$$e_R(t) \cdot \sum_{t' > t} s_Q(t')$$

tuple combinations arise from that. Alternatively, one could consider those tuples in Q that start at t. They join with all tuples in R that have ended before t, i.e.

$$s_Q(t) \cdot \sum_{t'<t} e_R(t')$$

tuple combinations arise from that. As above and for the same reasons we can concentrate on those t that are start- and endpoints. Thus the result size of a before join is

$$|R \overset{\text{bef}}{\bowtie} Q| = \sum_{j=1}^{N-1} \left(e_R(t_j) \cdot \sum_{l=j+1}^{N} s_Q(t_l) \right) = \sum_{j=2}^{N} \left(s_Q(t_j) \cdot \sum_{l=1}^{j-1} e_R(t_l) \right)$$

As the after join is an inverted before join, its result size is derived similarly as

$$|R \overset{\text{aft}}{\bowtie} Q| = \sum_{j=2}^{N} \left(s_R(t_j) \cdot \sum_{l=1}^{j-1} e_Q(t_l) \right) = \sum_{j=1}^{N-1} \left(e_Q(t_j) \cdot \sum_{l=j+1}^{N} s_R(t_l) \right)$$

Finally, a left-overlap join $R \overset{\text{lo}}{\bowtie} Q$ requires an r's timestamp's startpoint to lie inside the timestamp of a q if they are to qualify for the result. At a timepoint t these are $s_R(t) \cdot o_Q(t)$ tuples. Similarly, a right-overlap join $R \overset{\text{ro}}{\bowtie} Q$ requires an r's timestamp's endpoint to lie inside the timestamp of a q if they are to qualify for the result. At a timepoint t these are $e_R(t) \cdot o_Q(t)$ tuples. Again, we can concentrate on the t_j and calculate the result sizes of left-overlap- and right-overlap joins as

$$|R \overset{\text{lo}}{\bowtie} Q| = \sum_{j=1}^{N} s_R(t_j) \cdot o_Q(t_j)$$

$$|R \overset{\text{ro}}{\bowtie} Q| = \sum_{j=1}^{N} e_R(t_j) \cdot o_Q(t_j)$$

11.3.2 Composite Joins

We now turn to the calculation of result sizes of composite temporal joins. Actually, one can concentrate on showing how the intersections and unions of two (elementary, and later also composite) join results translate into formulas for calculating the respective result sizes. With composite joins we will have to rely on estimations rather than exact values because we cannot assume to the existence of IP-tables for the join results. Unfortunately, they cannot be calculated from the initial IP-tables and would have to be created by scanning the join result which is too expensive.

First, we discuss the *intersection* of two joins, as in the case of an equal join $R \overset{=}{\bowtie} Q$. As stated in table 11.2, it can be considered as an intersection of a start- and a finish join. We take the view that a selectivity factor of a join $R \bowtie_C Q$ gives the probability with which a tuple combination $r \circ q$ satisfies C. If $r \circ q$ satisfies the start join condition with a probability of *sel*$(R \overset{\text{sta}}{\bowtie} Q)$ and the finish join condition with a probability of *sel*$(R \overset{\text{fin}}{\bowtie} Q)$ then it qualifies with a probability of

$$\textit{sel}(R \overset{=}{\bowtie} Q) \;\approx\; \textit{sel}(R \overset{\text{sta}}{\bowtie} Q) \cdot \textit{sel}(R \overset{\text{fin}}{\bowtie} Q) \tag{11.2}$$

for the equal join $R \overset{=}{\bowtie} Q$ according the multiplication rule for independent probabilities [Bronstein and Semendjajew, 1987]. In order to emphasise the fact that this step is an approximation rather than an exact analytical result we use the $\approx$ symbol in (11.2). In fact, (11.2) requires some careful consideration: *sel*$(R \overset{\text{sta}}{\bowtie} Q)$ and *sel*$(R \overset{\text{fin}}{\bowtie} Q)$ are independent if and only if a tuple combination $r \circ q$ satisfies the start join condition irrespective from the question whether it satisfies the finish join condition. In other words, the interval startpoints must follow a probability distribution as well as the intervals' lengths and therefore the endpoints[1]. If this is the case, then (11.2) means that the result size of the equal join is

$$\begin{aligned}
|R \overset{=}{\bowtie} Q| \quad &= \quad \textit{sel}(R \overset{=}{\bowtie} Q) \cdot |R| \cdot |Q| \\
&\overset{(11.2)}{\approx} \quad \textit{sel}(R \overset{\text{sta}}{\bowtie} Q) \cdot \textit{sel}(R \overset{\text{fin}}{\bowtie} Q) \cdot |R| \cdot |Q| \\
&\overset{(11.1)}{=} \quad \frac{|R \overset{\text{sta}}{\bowtie} Q|}{|R| \cdot |Q|} \cdot \frac{|R \overset{\text{fin}}{\bowtie} Q|}{|R| \cdot |Q|} \cdot |R| \cdot |Q| \\
&= \quad \frac{|R \overset{\text{sta}}{\bowtie} Q| \cdot |R \overset{\text{fin}}{\bowtie} Q|}{|R| \cdot |Q|}
\end{aligned}$$

This is only an instance of the more general formula

$$|R \bowtie_C Q \cap R \bowtie_D Q| \;\approx\; \frac{|R \bowtie_C Q| \cdot |R \bowtie_D Q|}{|R| \cdot |Q|} \tag{11.3}$$

if the join selectivities of the two joins are independent from each other as outlined above.

Now we look at the *union* of two joins as in the case of an overlap join $R \overset{\text{olp}}{\bowtie} Q$. As shown in table 11.2, it can be regarded as the union of a left-overlap

[1] In the context of the phone calls scenario this would mean that the time when a phone call starts should not imply a certain length of the phone call.

join $R \overset{\text{lo}}{\bowtie} Q$ and a right-overlap join $R \overset{\text{ro}}{\bowtie} Q$. This case can be treated by the rule from set theory for calculating the size of the union of two sets $\mathcal{X}$ and $\mathcal{Y}$:

$$|\mathcal{X} \cup \mathcal{Y}| \;=\; |\mathcal{X}| + |\mathcal{Y}| - |\mathcal{X} \cap \mathcal{Y}|$$

This translates into

$$\begin{aligned} |R \bowtie_C Q \cup R \bowtie_D Q| \quad &= \quad |R \bowtie_C Q| + |R \bowtie_D Q| - |R \bowtie_C Q \cap R \bowtie_D Q| \\ &\overset{(11.3)}{\approx} \quad |R \bowtie_C Q| + |R \bowtie_D Q| - \frac{|R \bowtie_C Q| \cdot |R \bowtie_D Q|}{|R| \cdot |Q|} \end{aligned} \tag{11.4}$$

An example for applying (11.4) is the overlap join. However, we cannot just add up numbers because some tuples might appear in both joins, in this case these are those combinations $r \circ q$ in which r's timestamp is contained inside q's timestamp. So we have to deduct the number of tuples falling in the intersection of the elementary joins. Thus the result size is

$$|R \overset{\text{olp}}{\bowtie} Q| \;=\; |R \overset{\text{lo}}{\bowtie} Q \cup R \overset{\text{ro}}{\bowtie} Q| \;=\; |R \overset{\text{lo}}{\bowtie} Q| + |R \overset{\text{ro}}{\bowtie} Q| - |R \overset{\text{lo}}{\bowtie} Q \cap R \overset{\text{ro}}{\bowtie} Q|$$

which reflects (11.4).

Using (11.3) and (11.4), we can now break down complex temporal join conditions into smaller ones until we have only a set of elementary ones which can be computed according to the formulas given in section 11.3.1. This can be, for example, applied to the contain, during and intersection joins.

The latter one, however, can alternatively be treated as one of the elementary ones as the IP-tables provide sufficient information for calculating its size exactly: consider a timepoint t with $s_R(t)$ tuple timestamps in R starting at t. Then these intervals intersect with exactly $i_Q(t)$ tuple timestamps of Q. Alternatively, one can start with tuples of Q: $s_Q(t)$ timestamps start at t and intersect with $i_R(t)$ tuple timestamps in R. Further alternatives involve the consideration of intervals' endpoints. For reasons mentioned above, one can concentrate on the t_j. All in all, we get the following equations for computing the result size of an intersection join:

$$\begin{aligned} |R \overset{\text{int}}{\bowtie} Q| \;&=\; \sum_{j=1}^{N} s_R(t_j) \cdot i_Q(t_j) + i_R(t_j) \cdot s_Q(t_j) \\ &=\; \sum_{j=1}^{N} e_R(t_j) \cdot i_Q(t_j) + i_R(t_j) \cdot e_Q(t_j) \end{aligned} \tag{11.5}$$

11.3.3 Parallel and Other Partitioned Joins

In this thesis, we have been focusing on temporal joins that are processed by symmetrically partitioning the participating relations in order to create a number of smaller and independent joins. This is based on the algebraic expression

$$R \bowtie_C Q \;=\; R_1 \bowtie_C Q_1 \cup \cdots \cup R_m \bowtie_C Q_m \tag{11.6}$$

where equally indexed fragments R_k and Q_k are created in such a way that they hold tuples whose timestamps can possibly join with each other ($k = 1, \ldots, m$). In the case of the temporal intersection join, for example, this means that an R_k holds those tuples whose timestamps intersect with a certain range of the timeline; the corresponding Q_k is created in the same way.

With respect to join selectivity, this creates a new problem: a fragment R_k represents a number of tuples of R that have been selected *according to certain rules*. Therefore, an R_k does not necessarily have the same statistical properties as R. Thus, if we want to estimate the join selectivity of a partial join $R_k \bowtie_C Q_k$ we cannot use statistical approaches such as in [Segev et al., 1993] because they assume the same statistical properties for *all* fragments. To see why this is not necessarily the case, consider again the phone calls example mentioned in sections 2.1, 7.3.1 and 9.1.4. A fragment R_k can, for example, hold phone calls made during Christmas time or during a period with a promotional offer of cheap rates or during any other type of period which causes a different consumer behaviour. This means that R_k is likely to have widely different properties than many other R_j with $j \neq k$ or R itself. However, one has to keep in mind that the partial joins in (11.6) are processed in parallel. Therefore it is the most expensive one among these that determines the overall performance. Thus, if there is at least one partial join whose result size "gets out of hand" for the reasons mentioned above then this translates into an immediate performance penalty. In order to avoid such a situation one requires a method that can cope with statistical properties that vary over time.

Our analytical approach can tackle this problem. Consider, for example, a partial temporal intersection join $R_k \overset{\text{int}}{\bowtie} Q_k$ where R_k and Q_k hold tuples from R and Q, respectively, whose timestamps intersect with a time period that runs from time x to time y. Let

$$\begin{aligned} j_x &= \min\{j : j \in \{1, \ldots, N\} \text{ and } t_j \geq x\} \\ j_y &= \max\{j : j \in \{1, \ldots, N\} \text{ and } t_j \leq y\} \end{aligned}$$

i.e. j_x and j_y are the indices of the start- or endpoints within the set $\{t_1, \ldots, t_N\}$

that are closest to x and y, respectively, but inside the range between x and y, i.e. $x \leq t_{j_x} \leq t_{j_y} \leq y$.

At the beginning, at time x, we have those tuples whose timestamps start before time x. There are $o_R(t_{j_x-1})$ and $o_Q(t_{j_x-1})$ of these in R_k and Q_k respectively. All their timestamps intersect. Thus all their combinations are in the result of $R_k \overset{\text{int}}{\bowtie} Q_k$. Furthermore and similar to the derivation of equation (11.5), there are those tuples in R_k whose timestamps start between x and y. These join with those tuples in Q_k that intersect the respective startpoint. Therefore the result size of the partial intersection join is given by

$$|R_k \overset{\text{int}}{\bowtie} Q_k| \;=\; o_R(t_{j_x-1}) \cdot o_Q(t_{j_x-1}) \;+ \sum_{j=j_x}^{j_y} s_R(t_j) \cdot i_Q(t_j) + i_R(t_j) \cdot s_Q(t_j)$$

11.4 Summary

In this chapter, we have shown an analytical way of calculating temporal join result sizes or – respectively – temporal join selectivities. To our knowledge, there has only been one paper discussing the selectivity estimation for temporal joins [Segev et al., 1993]. Its approach requires that the statistical process that creates the timestamps is either well understood or follows certain standard probability distributions such as the Poisson distribution for interval startpoints or the Erlang-n distribution for interval lengths. The first case is quite rare: imagine the example of the distribution and lengths of telephone calls which depend on many statistical processes that are influenced by holidays, pricing, marketing or TV campaigns and even the weather. It is difficult to incorporate all these effects into a thorough statistical model for a query optimiser. In the second case, the assumptions can be erroneous for the same reasons.

In contrast to that, our technique is based on the information stored in IP-tables. For a set of elementary temporal joins, exact result sizes can be computed (section 11.3.1). For cases of temporal joins that arise from a composition of the elementary join conditions we gave the formulas (11.3) and (11.4). These allow to derive the result sizes of composite temporal joins from those of the elementary joins that are involved (section 11.3.2). Finally, we also provided a way to calculate result sizes of partial temporal joins that occur in parallel join processing (section 11.3.3).

The advantages of our analytical approach as opposed to statistical ones are

- Most results are *exact* rather than estimations.

- The calculations consider the fact that timestamps are often the result of a *variety* of *interfering* statistical processes.
- They are also sensitive to the fact that these statistical processes can *change over time* (see phone calls example). This property, for example, allowed to derive the result sizes of partial joins in parallel processing.
- It can be applied to *all* types of temporal data, regardless of the underlying semantics and its implications for statistical modeling. This means that one does not have to analyse the nature of the temporal data and the underlying statistical processes in order to be able to estimate result sizes but can work on a purely analytical basis regardless of the origin of the temporal data.

Chapter 12

Summary, Conclusions and Future Work

12.1 Summary

We now give a short summary of the main issues that have been discussed in this thesis.

In chapter 2, we motivated the significance and importance of interval data, especially in the context of temporal databases. The usage of interval data does not work well with many traditional performance-enhancing methods, such as indexes, or performance-critical algorithms, such as those that are traditionally used for the join operation. The latter were the focus of chapter 3 which gave an overview over the huge number of algorithms that have been designed and proposed in the past. Many of these are tuned to perform well with equi-join conditions, as these are the most frequent ones in conventional database processing. In chapter 4, we analysed how these algorithms can be adapted to process temporal joins. Those join algorithms that are not based on explicit and symmetric partitioning hardly required any changes. Hash and parallel join processing have gained an increased significance with the advent of parallel database systems in recent years, especially in the context of data warehousing and data mining. However, these techniques are based on explicitly and symmetrically partitioning the relations that participate in a join. Partitioning interval data is different from partitioning atomic data in the sense that it results in *non-disjoint* relation fragments due to intervals that overlap the partition's breakpoints. This can cause three types of overhead:

- a replication overhead,
- a processing overhead, and

- a duplicates overhead

In a first step, we adapted the hash and parallel join technique which now avoids the duplicates overhead and reduces the other two. We suspected that there is a major potential of further improving a partitioned temporal join's performance by carefully choosing the partition and thereby reducing the number of replicated tuples. In chapter 5, we looked at this issue from a theoretical point of view. The interval partitioning (IP) problem was defined and its complexity was analysed. The result was that there is an algorithm that computes an optimal solution in polynomial time. We could also relate IP to the sequential graph partitioning (SGP) problem. This confirmed the initial result and enables us in the future to take advantage of the many theoretical and practical results that have been obtained in the context of graph partitioning. The analytical part of the thesis was concluded in chapter 6 and the results were merged into the design of a process for the optimisation of partitioned temporal joins which consists of four stages (see figure 6.1):

- a *data analysis*, in which the temporal data is analysed in order to derive the characteristics of its timestamp intervals,
- a *synthesis of partitions*, in which several partitioning strategies create several partition candidates for the scenario,
- an *analysis of partitions*, in which the partition candidates are analysed with respect to their performance impact, and
- an *optimisation decision* that decides on the most convenient partition for processing the respective temporal join.

These stages were elaborated in the following chapters.

In chapter 7, IP-tables were introduced as a new type of metadata-structure. The information that is stored in an IP-table allows us to determine many parameters that influence the processing performance. We looked at several issues that concern the usage of IP-tables, such as their sizes and measures by which they can be decreased, the process of merging two or more IP-tables into one and finally how the information in the IP-tables can be maintained.

In chapter 8, we created a performance model for temporal join processing. This was divided into three steps. First, we created a model of the hardware architectures on which parallel and sequential database systems might run. It had to be general enough to embrace the large number of differing architectures that have been proposed and introduced. In a second step, we described

the model in which a temporal join is processed on top of the architectural model. Finally, we were able to create a very detailed cost model for processing a partitioned temporal join in parallel or sequentially. It takes advantage of the data that is provided by the IP-tables of the participating relations.

In chapter 9, three families of partitioning strategies were described: the uniform, underflow and minimum-overlaps strategies. All of them can be efficiently implemented on the base of IP-tables. Each strategy emphasises a certain goal, such as simplicity or reducing one or more performance-critical parameters. There are many possible variations of these strategies. We presented one such possibility which is based on preprocessing the IP-tables that are involved in the partitioning process and cutting out possibly bad breakpoint candidates (black-out preprocessing).

In chapter 10, a thorough evaluation of the optimisation process was provided. A parallel and a single-processor architecture were used. The experiments indicated advantages and disadvantages of the partitioning strategies and also gave useful information on how to choose suitable values for the input parameters of the strategies.

Finally, in chapter 11, we showed that IP-tables not only suit for partitioning purposes but have a much wider scope. They can be used to exactly calculate or, at least, to estimate the sizes of temporal join result. Such information is required by many query optimising modules for taking optimisation decisions. The main advantage of the IP-table based approach is that it does not require a deep insight into the, possibly complex, statistical properties of the underlying temporal application. Such an insight is necessary for the selectivity estimation methods that have previously been proposed.

12.2 Conclusions

The principal contribution of this thesis is the elaboration and description of a novel way in which partitioned temporal join processing can be optimised. All parts of the optimisation process can be made very efficient by using IP-tables. If the algorithms, e.g. those presented in chapter 9, had to be implemented on top of a data sampling approach then they would be very inefficient as most of them would need to scan the data sample various times. The theoretical and experimental results provide a base as to how an optimisation module of a database management system can be enhanced to cope with partitioned temporal joins.

Apart from this major contribution there is a list of further important results that were obtained when investigating various aspects of this work. They are the following:

- We designed a temporal hash algorithm that (a) avoids the duplicates overhead, (b) reduces replication by avoiding unnecessary tuple comparisons and (c) increases the opportunities for main memory join processing (see section 4.4.3).

- We showed that the interval partitioning (IP) problem has a polynomial solution. Furthermore, it is related to graph partitioning which is a well investigated problem.

- The IP-table has emerged from the analysis of the IP problem. It proved to be a very versatile metadata-structure that can describe the characteristics of the timestamp intervals that are found in one or more temporal relations. We used IP-tables mainly for interval partitioning purposes but showed that there is a wider scope for them: they can be used for estimating the selectivity of temporal join conditions too.

- The IP-table based selectivity estimation has proved to be more generally applicable than statistical methods that have been proposed in the past. The latter require a thorough understanding of the, possibly complex, statistical processes that underly the temporal application. Our analytical approach does not.

- The condensation of IP-tables is a very efficient way of reducing the sizes of the IP-tables, thereby accelerating the optimisation process. The experimental results on the impact of condensation on the quality of partitioning were very encouraging. Condensation factors a between 10 and 20 are feasible and hardly penalise the processing performance. This range of condensation factor values reduces an IP-table's size to below 0.5% of the corresponding relation's size (see table 7.1). This is far below the sizes of data samples that constitute an alternative to an IP-table based partitioning approach.

- Naive, uniform partitioning results in very poor processing performances for parallel temporal joins. Costs can be up to three times higher than with more sophisticated techniques, such as the underflow and minimum-overlaps strategies. See figure 10.7 or table 10.6, for example.

- On the single-processor machine, uniform partitioning proved to be a viable option as long as a large number m of breakpoints was chosen (see results in section 10.3).

- For parallel temporal join processing, the experimental results show that a good load balance – even at the expense of an increased number of replicated tuples – is far more important than minimising replication. This can be seen from the fact that the primary underflow strategy produced better performances than the primary minimum-overlaps strategy in most situations on the parallel architecture. This result is contrary to our initial assumption. However, this is an encouraging result in the sense that partitioning over an interval attribute is not that severely penalised and thus can be an alternative to partitioning over an atomic attribute, e.g. if the latter's values are heavily skewed and would therefore cause a severe load imbalance.

However, there are several issues about this work which require careful consideration and possibly some more research in the future. One of these issues is the efficiency of the maintenance of the IP-tables. In section 7.4, we were concerned with showing how IP-tables can be updated. Thus there was an emphasis on feasibility rather than efficiency. Therefore the algorithms in that section do not claim to be the most efficient ones. In fact, one could imagine temporal database applications and query situations in which the overhead that is imposed by IP-table updates might become so significant that it outweighs the benefits of the IP-tables. It is still unclear, for example, whether an operational database with frequent updates to its (temporal) tables would significantly suffer from the IP-table overhead. Some more analysis in this quantitative aspect is required, either to discard this possibility or to assume that such situations might appear. In the latter case, one might want to find indicators that identify such problematic situations.

Our analytical cost model is a further issue which needs some validation. In the past, similar approaches have proved to be valuable for qualitative analysis, e.g. in [Hua et al., 1991]. However, one cannot say the same about the quantitative aspect. Modern hardware, especially parallel machines, have become systems that employ many complex performance enhancing mechanisms (such as caching or special devices to accelerate broadcasts or other typical communication patterns over the interconnect) which we could hardly incorporate into our cost model if we wanted to keep it reasonably general

(to allow to derive conclusions for a wide range of platforms) and reasonably simple so that it could be efficiently used in a query optimiser. In other words: there is a good justification that if our cost model shows that strategy X performs better than strategy Y then this effect can be observed on a wide range of implementations. However, we still need some validation for the absolute numbers, i.e. if a strategy causes costs X according to our cost model then it remains to be seen how realistic this prediction is. But this could be confirmed by implementing the strategies and the join algorithm on real hardware or at least by simulating them using one of the available simulation tools.

Finally there is another issue that has to be considered carefully: the costs for the optimisation itself. We gave results of elapsed times for deriving the costs imposed by the various strategies. These elapsed times were obtained on a specific machine. For an optimiser it could be beneficial to have a cost model for the optimisation process of section 6.1 itself. In that way, it could decide whether it is worth while to consider expensive partitioning techniques, such as those of the minimum-overlaps family, or whether simple and fast ones are sufficient in the light of saving optimisation costs.

12.3 Future Work

As we have seen in the conclusions section, there are several possibilities to confirm and extend the applicability of this work. For example, one has to consider that many temporal join conditions do not only consist of an intersection, contain, overlap or during predicate between timestamp intervals but possibly also of additional non-temporal expressions, e.g. equality of (non-temporal) attribute values. Such a situation suggests that partitioning over the attributes that are involved in the equality condition should be the preferred option as there is no overhead imposed through tuple replication. However, if one of the equi-condition attributes holds heavily skewed values an optimiser might dismiss this option. As we have seen in the experiments, tuple replication has not as much impact as we initially expected. On a parallel architecture the predominant goal must be to achieve a good load balance. Therefore, partitioning over the timestamp intervals is still a feasible alternative to partitioning over equi-condition attributes in the same way as the fragment-and-replicate technique has proved to be a valuable alternative to symmetric partitioning in commercial parallel query processing [Tseng and Reiner, 1993], despite the overhead that it incurs. It is necessary to get some experimental results on the

issue when a query optimiser should opt for partitioning over interval timestamps.

A second issue that could contribute to the appeal of IP-tables is to resolve the doubts about whether IP-tables can be maintained in an efficient way. There are various alternatives if IP-table maintenance becomes an efficiency problem and future research could analyse these alternatives:

- We have already seen that condensation is a good possibility to decrease the sizes of IP-tables without doing a lot of harm to the quality of the optimisation process. Reducing the sizes of the IP-tables should have an immediate performance benefit also for the IP-table update operations. One would need to know whether condensation is sufficient in the cases in which IP-table maintenance becomes a problem.

- As mentioned in section 7.5, there might not be many individual updates to temporal relations in a data warehouse environment but one bulk update, for example once per night. In this case, one could compute a temporary IP-table for the bulk update (which should be significantly more efficient than updating an existing IP-table) and then merge this temporary IP-table with the existing IP-table of the corresponding temporal relation.

- One could argue that condensation proved that we do not require exact numbers from the IP-tables. Therefore, one could consider that IP-tables do not require immediate updates. The latter could be accumulated and be processed similarly to the bulk update in a data warehouse or one could simply recompute a temporal relation's IP-table every now and then. One would need some experimental results in order to see if such an approach is viable.

- Finally, one could look at more efficient algorithms for the IP-table updates than those that we presented in section 7.4. One possibility could be to store the values $s_R(t)$ and $e_R(t)$ rather than $s_R(t)$ and $o_R(t)$. This makes the IP-table update operations more efficient (the for-loops in figures 7.10 – 7.15 can be avoided) but imposes more work when using IP-tables (one has to use the recursive equations in figure 5.1(a) rather than the non-recursive ones of figure 5.1(b)). One would need to determine the trade-off between these two effects.

A discussion of these options along with a quantitative analysis of the impact of the update operations is necessary and certainly an issue for future research.

This might be supported by findings made in the context of histograms as outlined in section 7.6.

The validation of our cost model by implementation of simulation is another imminent task that should be tackled in future research. It could be done in two stages. The first one would try to confirm the relative differences between partitioning strategies. This would consolidate many statements made in this thesis (e.g. the statement that uniform partitioning can be up to three times more expensive on a parallel machine than underflow partitioning). In a second stage, one would try to validate the absolute numbers that we obtained from the cost model. As outlined in section 6.2, it would be especially useful to bring our cost model in line with cost models for other join techniques in order to allow an optimiser to select the most efficient join algorithm.

As we also mentioned in the conclusions, it would be advantageous to have a cost model for the optimisation process itself. However, considering the wide range of possible partitioning strategies and also the wide range of possible implementations – figure 6.1, for example, suggests that there is a good chance to parallelise the optimisation too – makes this task costly and tedious but not impossible.

As we have seen in chapter 11, IP-tables prove to be a metadata-structure whose applicability goes beyond interval partitioning for join processing. Selectivity estimation is one area and we require experimental analysis of the results that were obtained in chapter 11. For example, one needs to investigate the impact of condensation on the quality of the selectivity results.

A further area to which IP-tables and interval partitioning is relevant is that of temporal index structures. Here, tree balancing is a major task in order to optimise memory requirements and access times for such indexes. In fact, Gunadhi and Segev met similar partitioning problems for temporal indexes as we did for temporal joins [Gunadhi and Segev, 1993]. We therefore expect that our IP-table based approach could be beneficial in that area too.

All this can establish IP-tables as a generally useful index structure for interval data. The initial results in this thesis are very encouraging in this respect and provide the base for future research.

Appendix A

Summary of the Cost Model

Hardware Parameters	
Parameter	**Description**
$\mathcal{M}$	number of processing nodes
$\mathcal{N}$	number of processors per node
$\mathcal{M} \cdot \mathcal{N}$	total number of processors
μ	processor speed in MIPS
mem	main memory available to the application per node
w_{io}	disk I/O bandwidth per node
w_{com}	communication bandwidth
w_{mem}	memory bandwidth per node
I_{proc}	number of CPU instructions for processing a tuple in each step
I_{hash}	number of CPU instructions for hashing a tuple
I_{com}	number of CPU instructions for initiating a data transfer
I_{io}	number of CPU instructions for initiating a disk I/O
b	page size

Table A.1: Hardware parameters.

Data Parameters	
Parameter	**Description**
$\|R\|$	number of tuples in R
$\|Q\|$	number of tuples in Q
$\|r\|$	size of a tuple $r \in R$ in bytes
$\|q\|$	size of a tuple $q \in Q$ in bytes
τ_R	average length of an interval in R in chronons
τ_Q	average length of an interval in Q in chronons

Table A.2: Data parameters.

Partition-related Parameters	
Parameter	**Description**
m	number of fragments, subjoins in (3.6) = number of segments into which the lifespan of the relation(s) is divided by a partition $\{p_1, \ldots, p_{m-1}\}$
$\|R'_k\|$	number of tuples in R'_k = number of intervals in R that start within $(p_{k-1}, p_k]$
$\|R''_k\|$	number of tuples in R''_k = number of intervals in R that intersect with but do not start in $(p_{k-1}, p_k]$
$\|Q'_k\|$	number of tuples in Q'_k = number of intervals in Q that start within $(p_{k-1}, p_k]$
$\|Q''_k\|$	number of tuples in Q''_k = number of intervals in Q that intersect with but do not start in $(p_{k-1}, p_k]$
δ_R	average number of segments with which an interval of R intersects
γ_R	average number of processing nodes at which a tuple of R has to reside after repartitioning
δ_Q	average number of segments with which an interval of Q intersects
γ_Q	average number of processing nodes at which a tuple of Q has to reside after repartitioning
λ_k	number of blocks into which R'_k is divided for a nested-block join computation of the subjoins $R'_k \bowtie_C Q''_k$ and $R'_k \bowtie_C Q'_k$
φ_k	number of blocks into which Q'_k is divided for a nested-block join computation of the subjoin $R''_k \bowtie_C Q'_k$

Table A.3: Partition related parameters.

Stage 1 (a)	
Disk I/O	$\frac{\lvert R\rvert}{\mathcal{M}} \cdot \frac{\lvert r\rvert}{w_{io}}$
Communication	
CPU	$\frac{\lvert R\rvert}{\mathcal{MN}} \cdot \frac{\lvert r\rvert}{b} \cdot \frac{I_{io}}{\mu}$
Memory	

Table A.4: Cost components for stage 1 (a).

Stage 1 (b)	
Disk I/O	
Communication	$\frac{\mathcal{M}-1}{\mathcal{M}} \cdot \lvert R\rvert \cdot \gamma_R \cdot \frac{\lvert r\rvert}{w_{com}}$
CPU	$\frac{\mathcal{M}-1}{\mathcal{M}} \cdot \frac{\lvert R\rvert}{\mathcal{MN}} \cdot \gamma_R \cdot \frac{I_{com}}{\mu} + \frac{\lvert R\rvert}{\mathcal{MN}} \cdot \frac{I_{hash}}{\mu}$
Memory	$\frac{\lvert R\rvert}{\mathcal{M}} \cdot \delta_R \cdot \frac{\lvert r\rvert}{w_{mem}}$

Table A.5: Cost components for stage 1 (b).

Stage 1 (c)	
Disk I/O	
Communication	
CPU	
Memory	$\frac{\lvert R\rvert}{\mathcal{M}} \cdot \frac{\lvert r\rvert}{w_{mem}} \cdot \min\{\delta_R, \frac{m}{\mathcal{M}}\}$

Table A.6: Cost components for stage 1 (c).

Stage 1 (d)	
Disk I/O	$\max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R'_k \rvert + \lvert R''_k \rvert \right\} \cdot \frac{\lvert r \rvert}{w_{io}}$
Communication	
CPU	$\max_{j=1}^{\mathcal{MN}} \left\{ \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R'_k \rvert + \lvert R''_k \rvert \right\} \cdot \frac{\lvert r \rvert}{b} \cdot \frac{I_{io}}{\mu}$
Memory	

Table A.7: Cost components for stage 1 (d).

Stage 2 (a)	
Disk I/O	$\frac{1}{w_{io}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R'_k \rvert \cdot \lvert r \rvert + \lvert Q''_k \rvert \cdot \lvert q \rvert \cdot \lambda_k \right\}$
CPU	$\max_{j=1}^{\mathcal{MN}} \left\{ \frac{1}{b} \cdot \frac{I_{io}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R'_k \rvert \cdot \lvert r \rvert + \lvert Q''_k \rvert \cdot \lvert q \rvert \cdot \lambda_k + \frac{I_{proc}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R'_k \rvert \cdot \lvert Q''_k \rvert \right\}$
Memory	$\frac{\lvert r \rvert}{w_{mem}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R'_k \rvert \cdot \lvert Q''_k \rvert \right\}$

Table A.8: Cost components for the joining stage 2 (a).

Stage 2 (b)	
Disk I/O	$\frac{1}{w_{io}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R'_k \rvert \cdot \lvert r \rvert \cdot \frac{\lambda_k - 1}{\lambda_k} + \lvert Q'_k \rvert \cdot \lvert q \rvert \cdot \lambda_k \right\}$
CPU	$\max_{j=1}^{\mathcal{MN}} \left\{ \frac{1}{b} \cdot \frac{I_{io}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R'_k \rvert \cdot \lvert r \rvert \cdot \frac{\lambda_k - 1}{\lambda_k} + \lvert Q'_k \rvert \cdot \lvert q \rvert \cdot \lambda_k + \frac{I_{proc}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R'_k \rvert \cdot \lvert Q'_k \rvert \right\}$
Memory	$\frac{\lvert r \rvert}{w_{mem}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R'_k \rvert \cdot \lvert Q'_k \rvert \right\}$

Table A.9: Cost components for the joining stage 2 (b).

Stage 2 (c)	
Disk I/O	$\frac{1}{w_{io}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R''_k \rvert \cdot \lvert r \rvert \cdot \varphi_k + \lvert Q'_k \rvert \cdot \lvert q \rvert \right\}$
CPU	$\max_{j=1}^{\mathcal{MN}} \left\{ \frac{1}{b} \cdot \frac{I_{io}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R''_k \rvert \cdot \lvert r \rvert \cdot \varphi_k + \lvert Q'_k \rvert \cdot \lvert q \rvert + \frac{I_{proc}}{\mu} \cdot \sum_{k=\textit{first}(j)}^{\textit{last}(j)} \lvert R''_k \rvert \cdot \lvert Q'_k \rvert \right\}$
Memory	$\frac{\lvert q \rvert}{w_{mem}} \cdot \max_{i=1}^{\mathcal{M}} \left\{ \sum_{k=\textit{first-node}(i)}^{\textit{last-node}(i)} \lvert R''_k \rvert \cdot \lvert Q'_k \rvert \right\}$

Table A.10: Cost components for the joining stage 2 (c).

Appendix B

Test Data Creation

Sections B.1 and B.2 respectively show the source code of the PERL programs that were used to convert login information obtained by UNIX's `last` command into the integer timestamps for relations R and Q which were used in chapter 10.

B.1 Timestamps for R

```
#!/usr/local/bin/perl

# ------------------------------------------------
# week-long lifespan, consider weekday (periodic)
# ------------------------------------------------
# This is a PERL script to convert the output of the "last" command.
# The login times are converted to minutes (eg. Wed, 9:24 ->
# 2*1024 + 9*60 + 24 = 2572). The result gives intervals within the
# lifespan 0..10079 (10080 = number of minutes within a week).


# an associative array to map weekday names to integers

%weekday = ("Mon",0,"Tue",1,"Wed",2,"Thu",3,"Fri",4,"Sat",5,"Sun",6);


# number of minutes per day

$one_day = 24 * 60;


# subsequently read lines of standard input ...

while (<>) {

   # split an input line using white spaces as separators

   @fields = split(/\s+/,$_);

   # convert hh:mm start and end times into integers
```

```
   ($hours,$minutes) = split(/:/,$fields[6]);
   $start = $weekday{$fields[3]} * $one_day + $hours * 60 + $minutes;
   ($hours,$minutes) = split(/:/,$fields[8]);
   $end   = $weekday{$fields[3]} * $one_day + $hours * 60 + $minutes;

   # deal with logins that ran over midnight

   if ($start > $end) {
      $end = $end + $one_day;
   }

   # write interval [$start,$end] to standard output;
   # if the [$start,$end] does not fall within the scope [0,10079]
   # then a "point" interval [$start,$start] is used

   if (($start <= $end) && ($end < 10080)) {
      print "[",$start,",",$end,"]\n";
   }
   else {
      print "[",$start,",",$start,"]\n";
   }
}
```

B.2 Timestamps for Q

```
#!/usr/local/bin/perl

# ------------------------------------------------------------
# week-long lifespan, do not consider weekday (non-periodic)
# ------------------------------------------------------------
# This is a PERL script to convert the output of the "last" command.
# Day information is negelected, only login start and end times are
# considered. The login times are converted to minutes (eg. 9:24 ->
# 9*60+24 = 524) and the projected from a daytime base to a weektime
# base (eg. 524 -> 524*7+<random number between 0 and 6>  = 3948..3954).
# The result gives intervals within the lifespan 0..10079
# (10080 = number of minutes within a week).


# number of minutes per day

$one_day = 24 * 60;


# initialise random generator

srand;


# subsequently read lines of standard input ...

while (<>) {
```

```
   # split an input line using white spaces as separators

   @fields = split(/\s+/,$_);

   # convert hh:mm start and end times into integers

   ($hours,$minutes) = split(/:/,$fields[6]);
   $start = $hours * 60 + $minutes;
   ($hours,$minutes) = split(/:/,$fields[8]);
   $end   = $hours * 60 + $minutes;

   # project daytime period to weektime period

   $start = $start * 7 + int(rand(7));
   $end   = $end   * 7 + int(rand(7));

   # deal with logins that ran over midnight

   if ($start > $end) {
      $end = $end + $one_day;
   }

   # write interval [$start,$end] to standard output;
   # if the [$start,$end] does not fall within the scope [0,10079]
   # then a "point" interval [$start,$start] is used

   if (($start <= $end) && ($end < 10080)) {
      print "[",$start,",",$end,"]\n";
   }
   else {
      print "[",$start,",",$start,"]\n";
   }
}
```

Appendix C

Manipulation of Interval Lengths

This is the C source code for a function *change_lengths (change)* that changes the average interval length τ by the value given in *change*.

```
/************************************/
/*****  change_lengths (change)  *****/
/************************************/
/*
   Changes the average length of the intervals by the
   value given in "change". The interval starpoints
   are stored in ts[0..N-1] and the corresponding
   endpoints in te[0..N-1].
   The function randomly picks an interval and adds
   or subtracts chronons in order to achieve the new
   average length.
*/

void change_lengths (int change)  {

/*
   Global variables or macros:
   ---------------------------
   N              = total number of intervals
   ts[0..N-1]     = intervals startpoints
   te[0..N-1]     = intervals endpoints
   tmax           = maximum value for a ts[] or a te[]
   MAX_IDLE       = max. number of loop runs that can be idle
   total_length   = sum of all intervals' lengths
   average_length = current average length of the intervals

   Local variables:
   ----------------
   i          = index of interval whose length will be changed
   useless    = counter for the number of intervals with ts >= te
   idle       = number of subsequent idle loop passes
   difference = interval i's length = te[i] - ts[i]
   direction  = +1  if average length is to be increased
              = -1  if average length is to be decreased
   to_change  = number of chronons that remain to be
                added or subtracted
   abs_change = abs(change) = change * direction
```

```
   end        = 0   if ts[i] is to be moved to the left or right
              = 1   if te[i] is to be moved to the left or right
*/

   unsigned i, useless, idle, difference, abs_change;
   int      end = 1;
   long     to_change;
   short    direction;


   /* Are chronons added or subtracted */

   if (change >= 0)
     direction = 1;
   else
     direction = -1;


   /* determine the absolute value of "change" */

   abs_change = change * direction;


   /* total amount of chronons to add / to substract */

   to_change = change * N;


   /* determine new total length */

   total_length = total_length + to_change;


   /* initialise random generator */

   srand(seed);


   /* initialise "idle" */

   idle = 0;


   /* main loop that subsequently adds or substracts chronons */

   while ((to_change > 0) && (idle < MAX_IDLE)) {

      /* find a suitable interval */

      do {
        i = rand() % N;
      } while ((ts[i]==0) && (te[i]==tmax));


      /* will start- or endpoint be changed? */
```

```
end = rand() % 2;


/* increase lengths */

if (direction == 1) {

  idle++;

  if ((end == 0) && (ts[i] > 0)) {
    ts[i]--;
    to_change--;
    idle = 0;
  }

  if ((end == 1) && (te[i] < tmax)) {
      te[i]++;
      to_change--;
      idle = 0;
    }
}

/* decrease lengths */

else {

  /* search for another interval if the current one
     comprises no chronon, i.e. it is a timepoint.
     The search is stopped when the number of useless
     intervals matches the total number of intervals. */

  useless = 0;

  while ((ts[i] == te[i]) && (useless < N)) {
    i = (i+1) % N;
    useless++;
  }

  /* If no suitable interval is found then leave the loop */

  if (useless >= N)
    idle = MAX_IDLE;

  /* If the interval i is suitable then ... */

  if (ts[i] < te[i]) {

    difference = te[i] - ts[i];

    /* decrease length as much as possible */

    if (difference <= abs_change) {
      if (end)
        te[i] = ts[i];
      else
        ts[i] = te[i];
```

```
            to_change = to_change + difference;
          }
          else if ((difference > abs_change) &&
                   (to_change  < (2*change))  )   {
            if (!end)
              ts[i]     = ts[i] + abs_change;
            else
              te[i]     = te[i] - abs_change;
            to_change = to_change + abs_change;
          }
          else {
            if (!end)
              ts[i]++;
            else
              te[i]--;
            to_change++;
          }
        }
      }

   }

   if (to_change != 0) {
     fprintf(stderr,"--- Warning ---\n");
     fprintf(stderr,"Too many idle attempts!\n");
     fprintf(stderr,"Abandoned change loop.\n");
     total_length = total_length - to_change;
   }

   /* determine new average length */

   average_length = total_length / N;


   return;
}
```

Appendix D

Profiles of the R_τ and Q_τ

The following figures respectively show the profiles $i_{R_\tau}(t)$ and $i_{Q_\tau}(t)$ of the relations R_τ and Q_τ as they were used for the experiments in section 10.5. Values for the average interval length τ are 200, 400, 600, 800, 1000 and 1200.

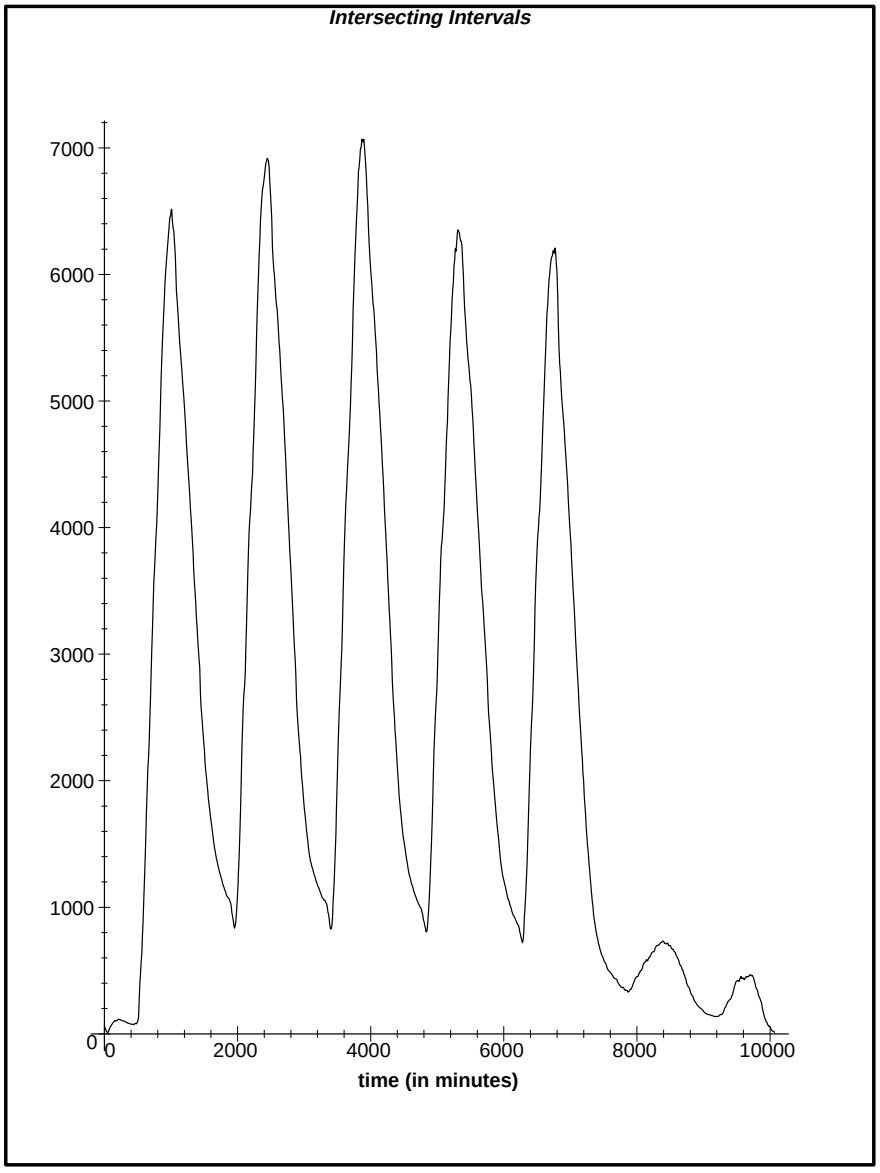


Figure D.1: The profile of R_τ with $\tau = 200$.

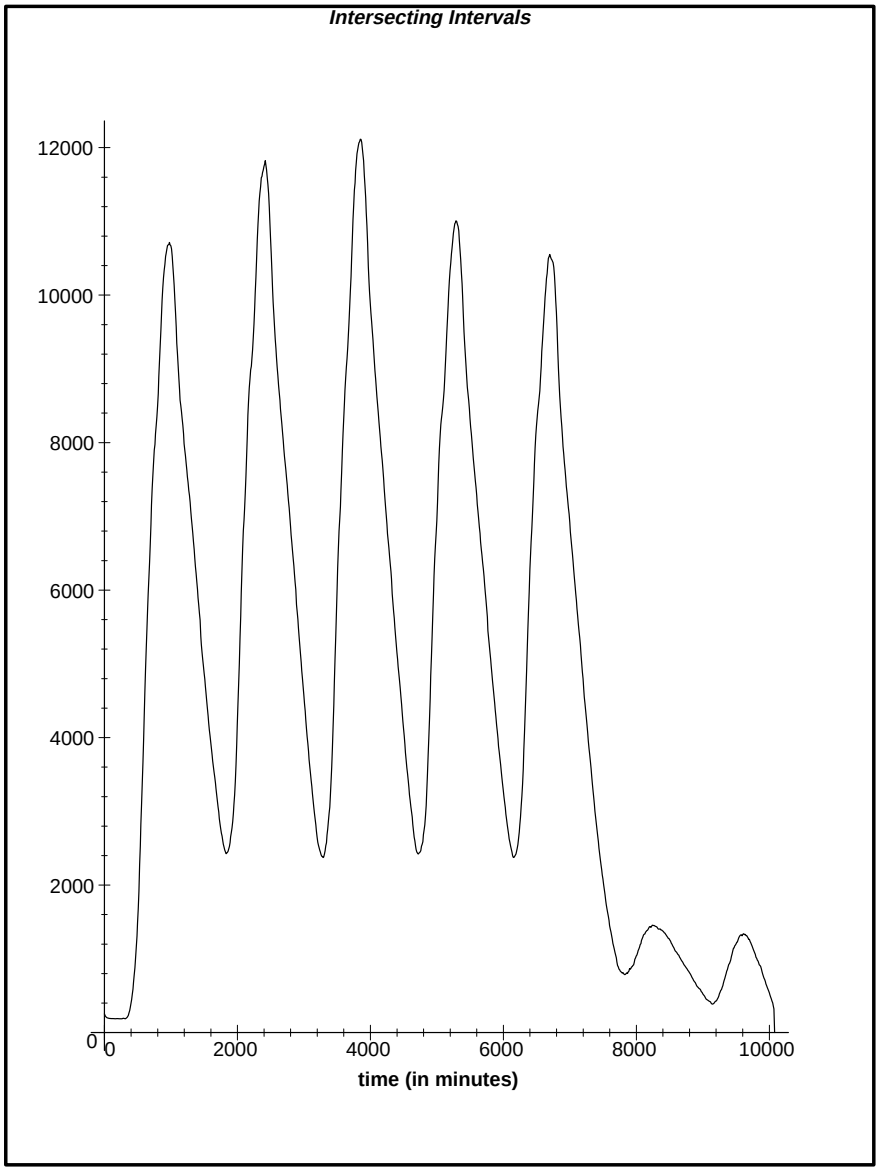


Figure D.2: The profile of R_τ with $\tau = 400$.

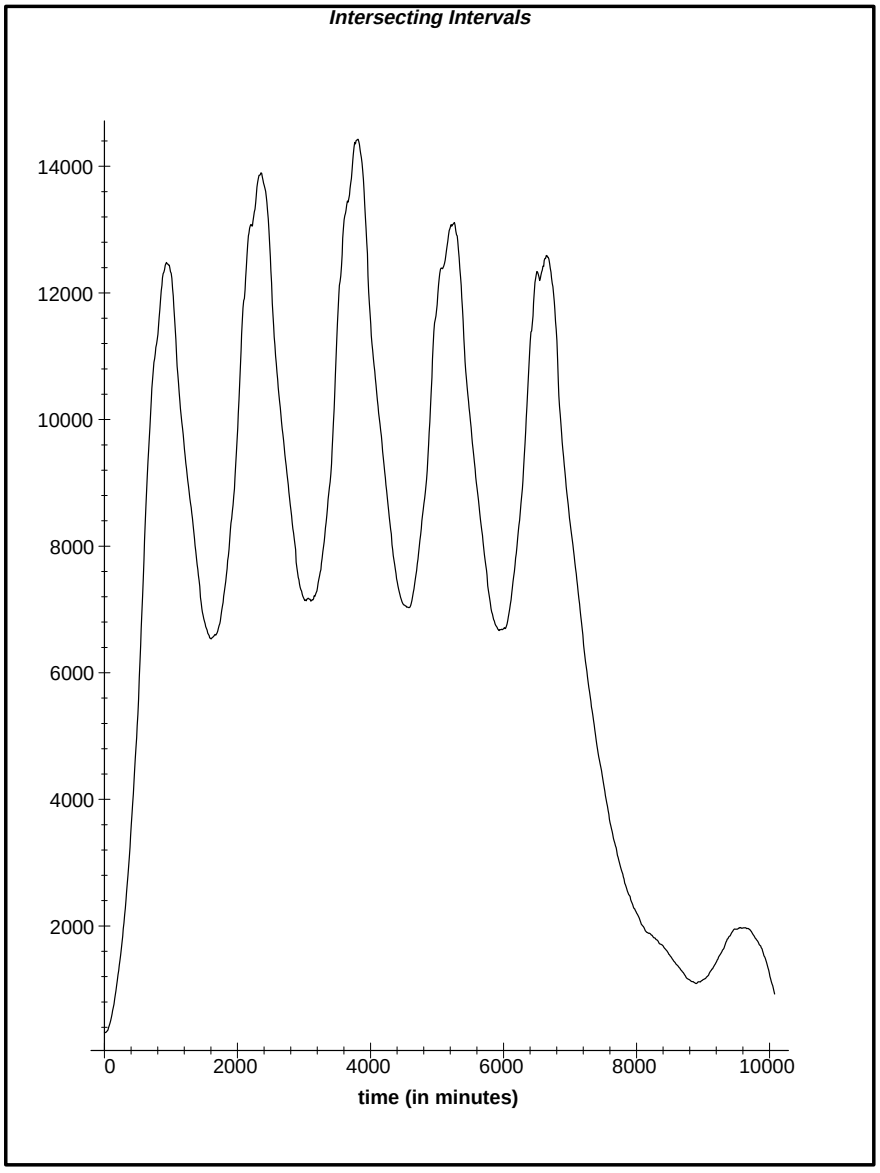


Figure D.3: The profile of R_τ with $\tau = 600$.

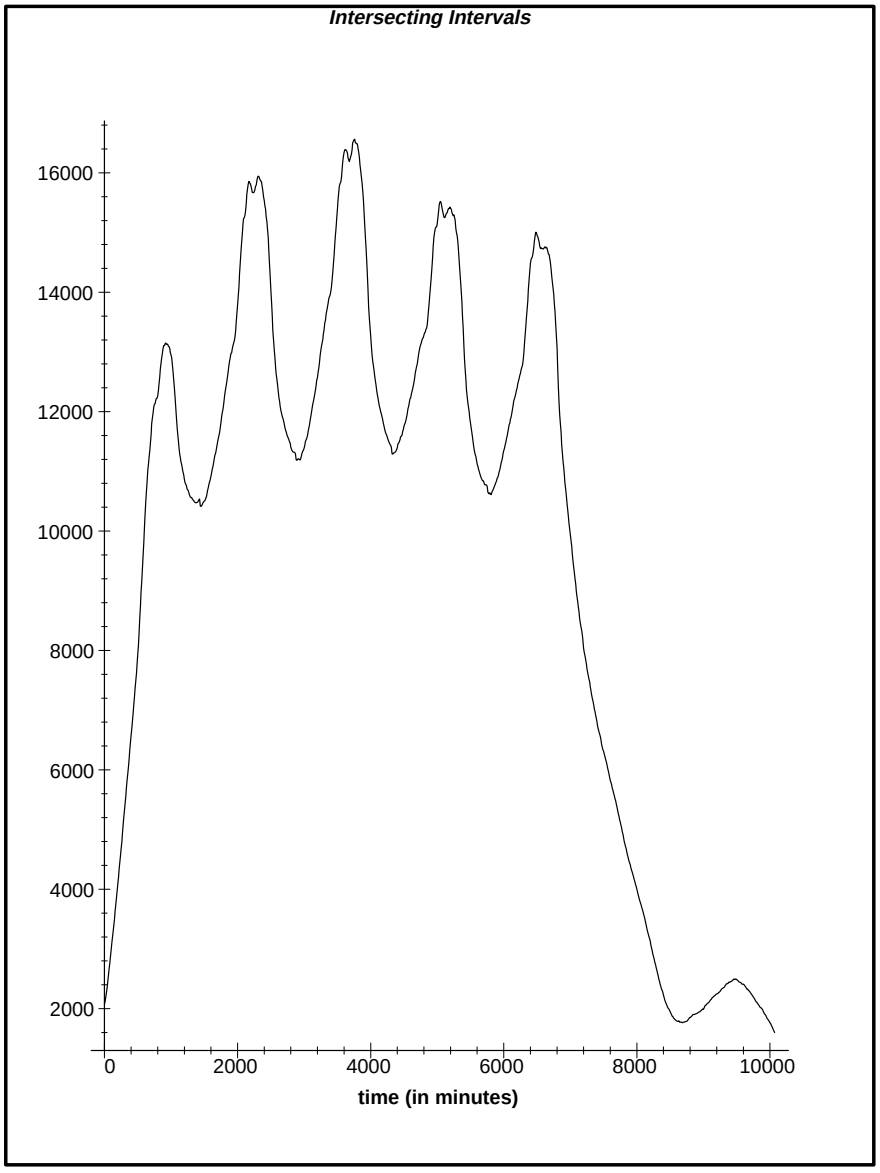


Figure D.4: The profile of R_τ with $\tau = 800$.

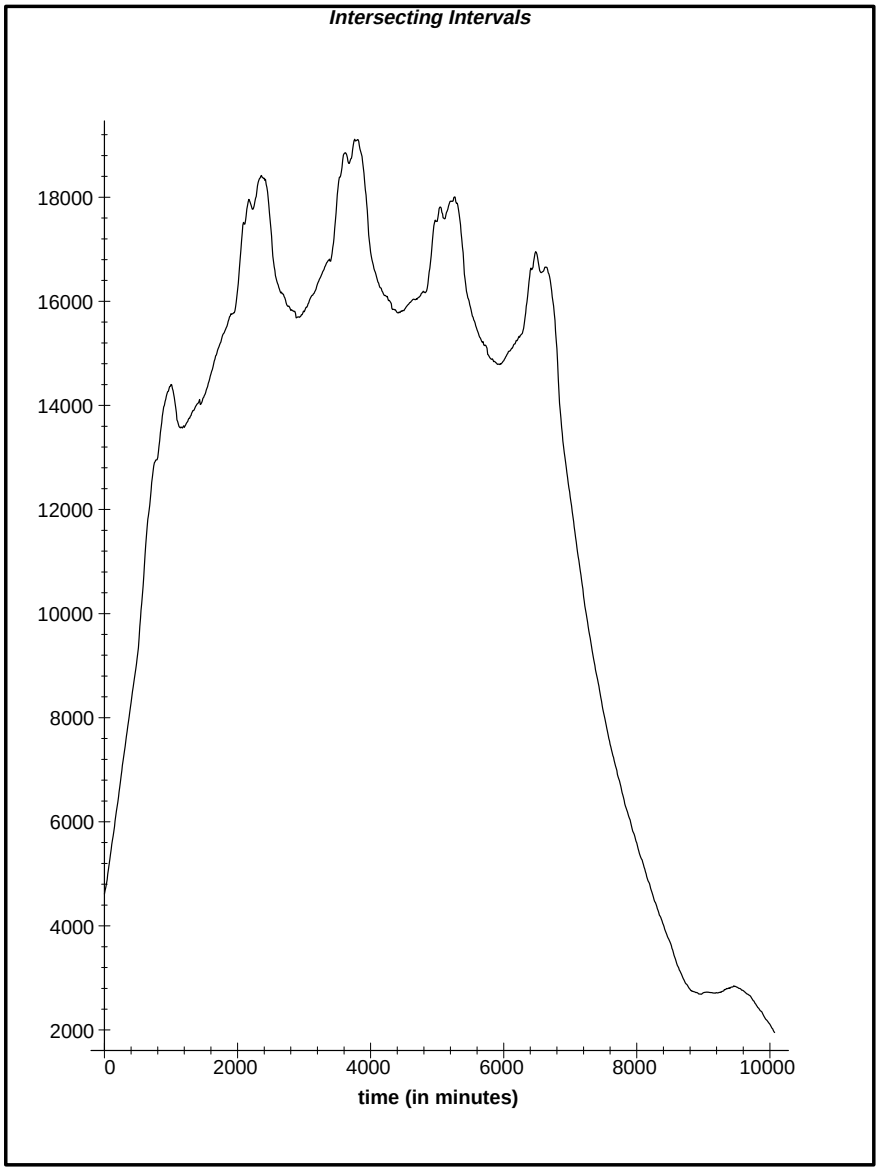


Figure D.5: The profile of R_τ with $\tau = 1000$.

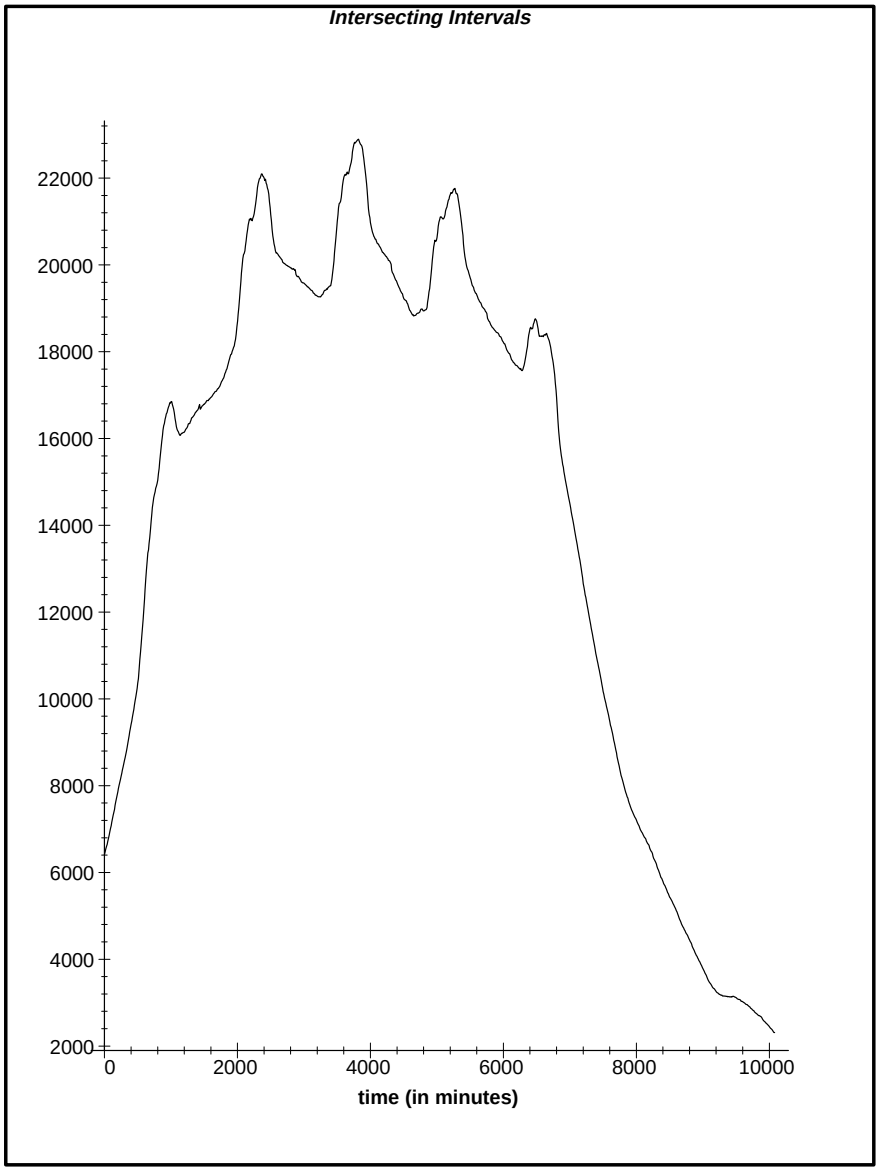


Figure D.6: The profile of R_τ with $\tau = 1200$.

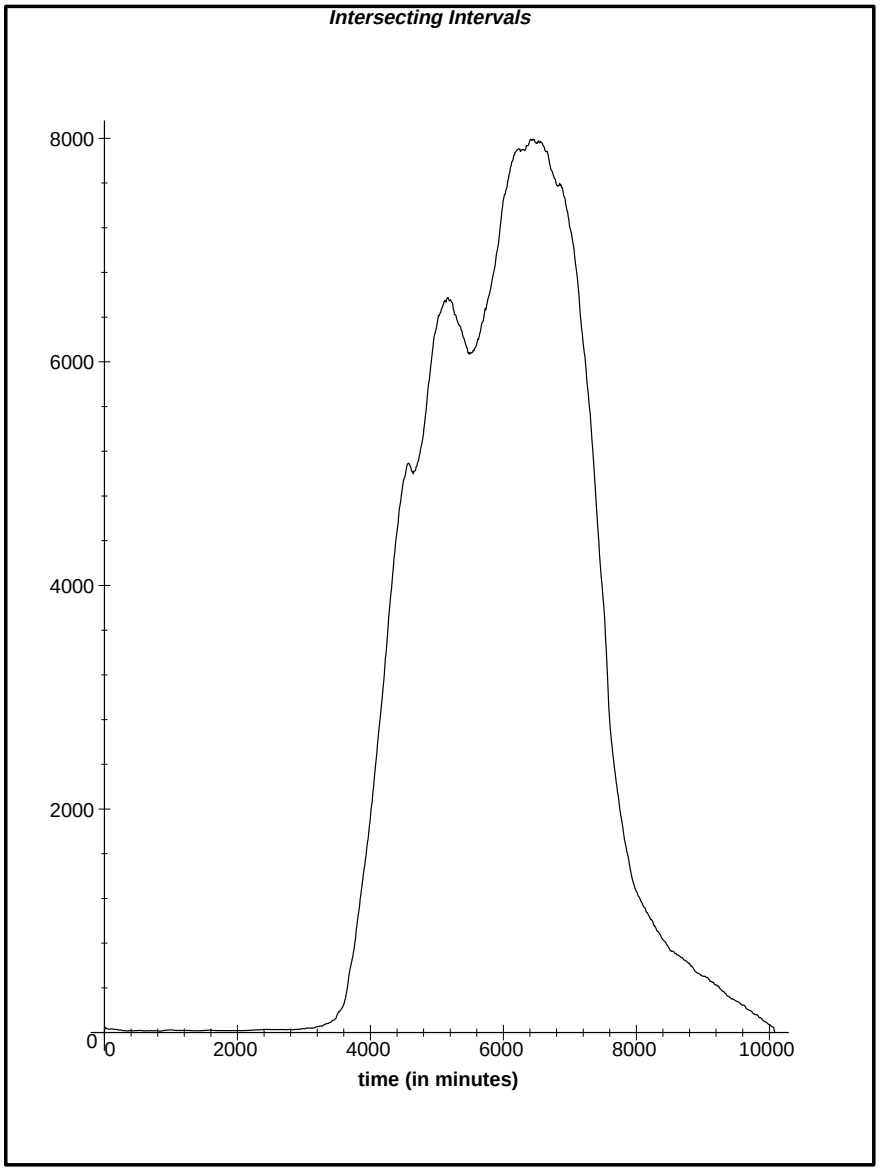


Figure D.7: The profile of Q_τ with $\tau = 200$.

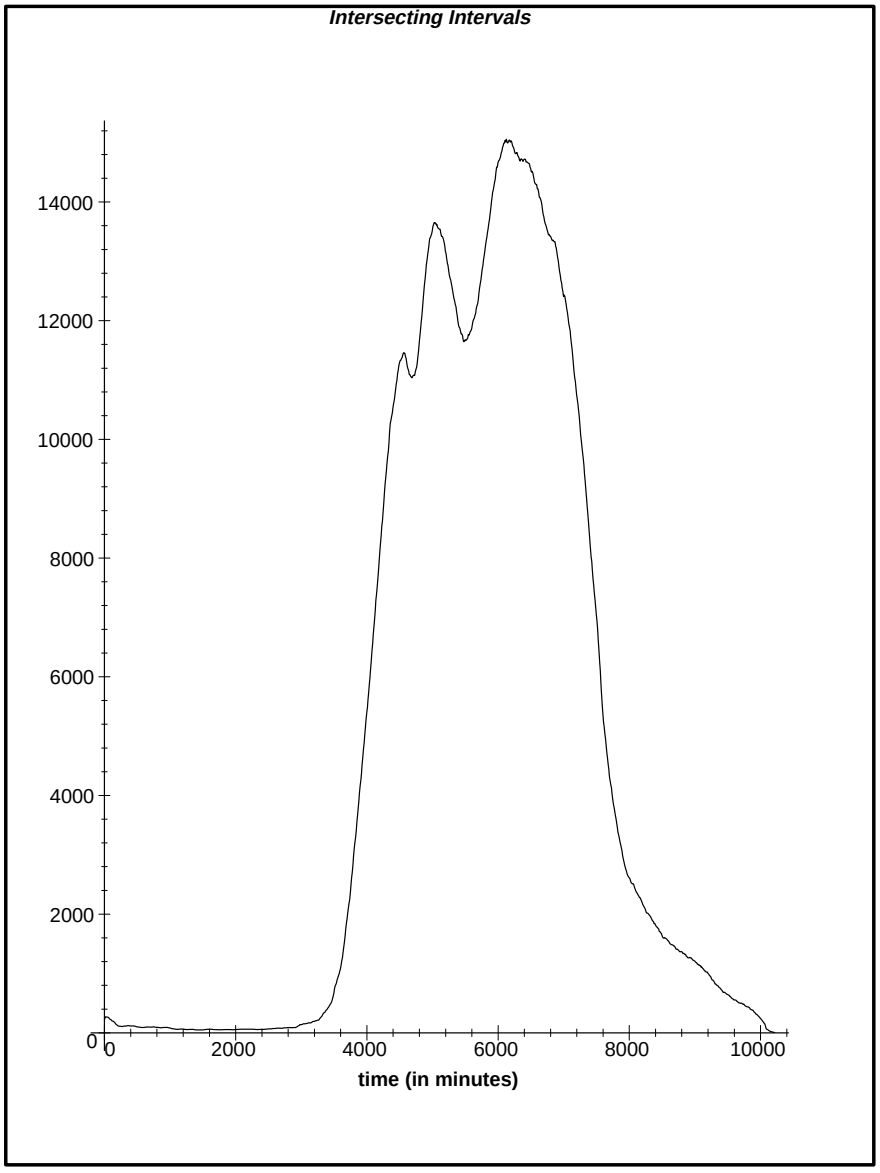


Figure D.8: The profile of Q_τ with $\tau = 400$.

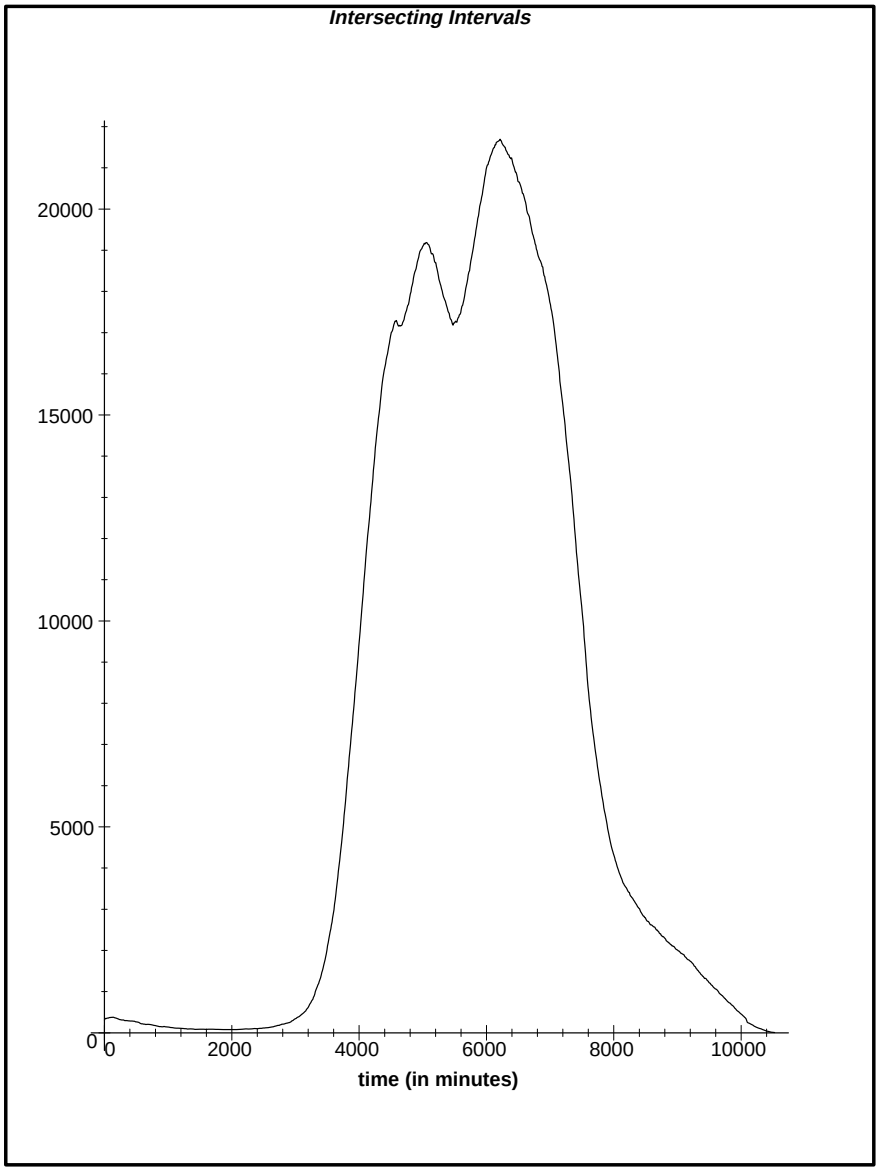


Figure D.9: The profile of Q_τ with $\tau = 600$.

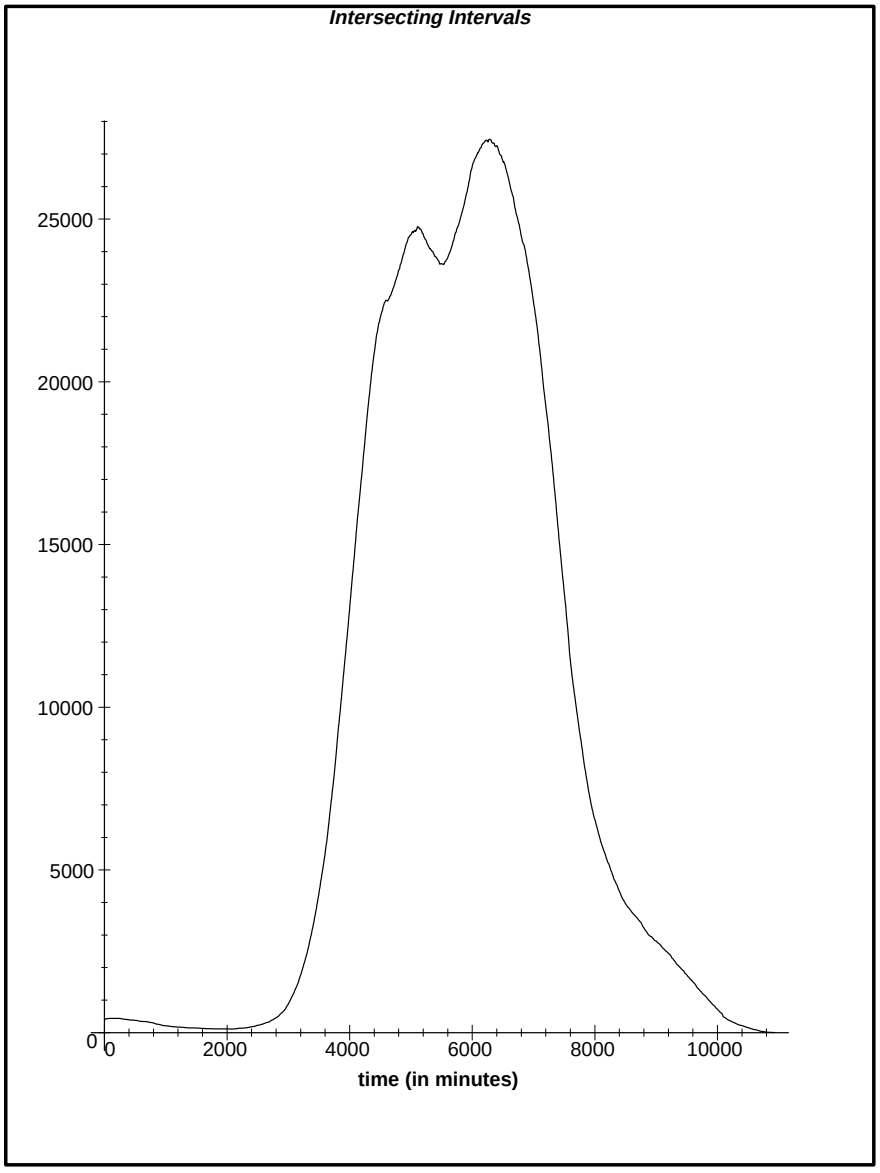


Figure D.10: The profile of Q_τ with $\tau = 800$.

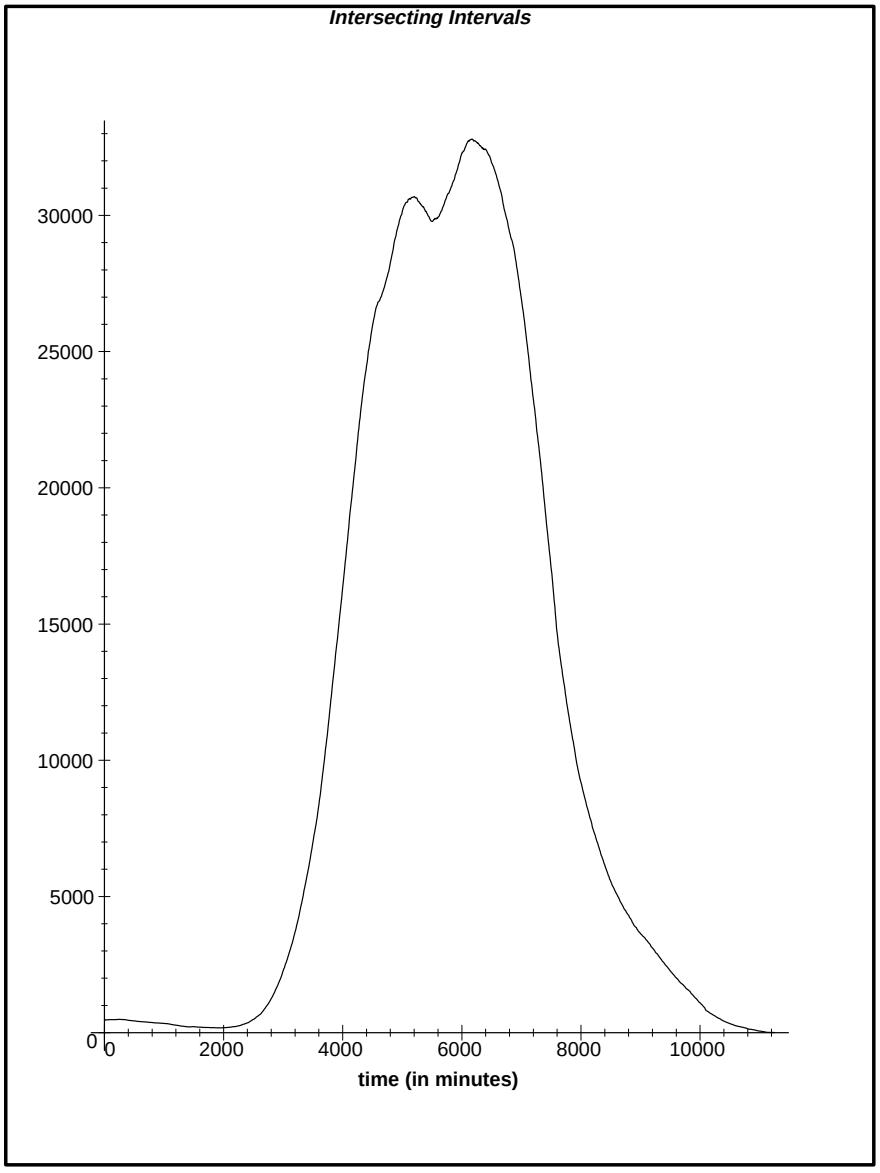


Figure D.11: The profile of Q_τ with $\tau = 1000$.

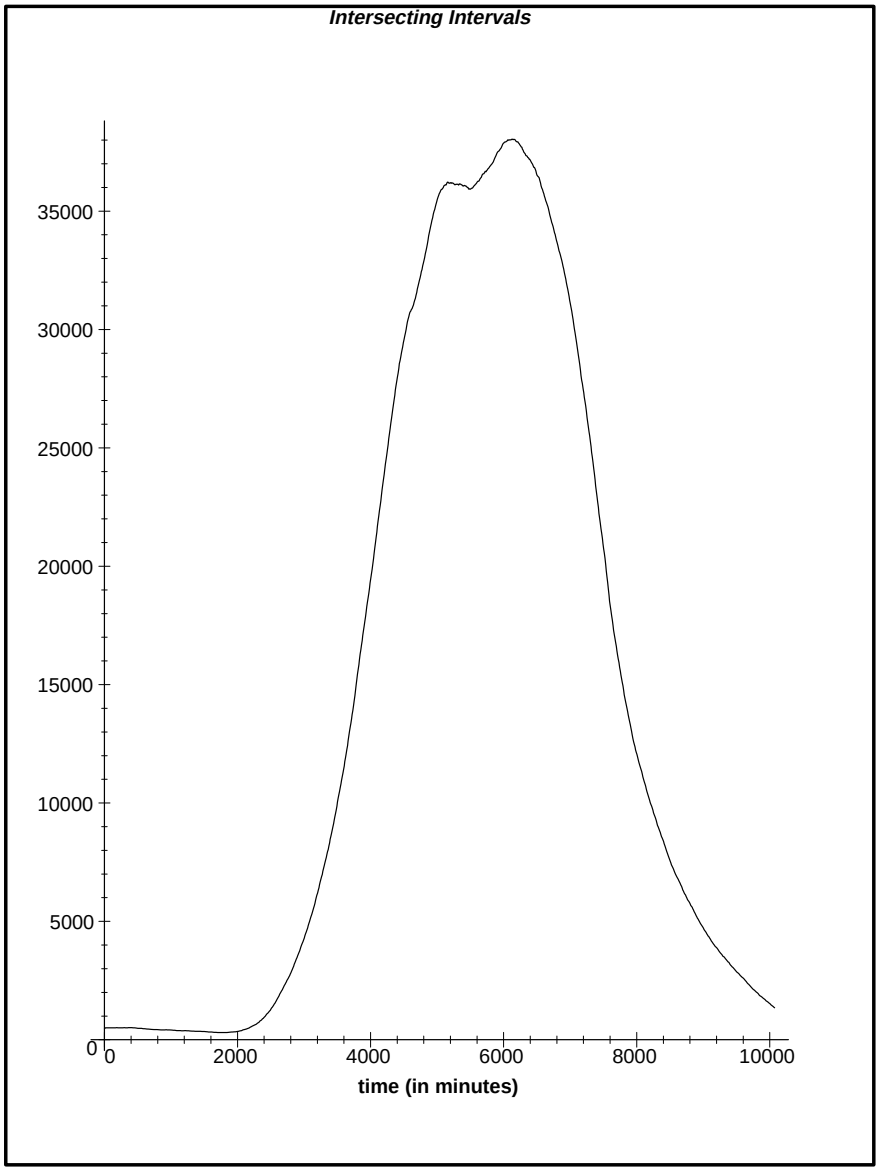


Figure D.12: The profile of Q_τ with $\tau = 1200$.

List of Figures

List of Tables

Bibliography

Allen, J. (1983). Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843.

Balcázar, J., Díaz, J., and Gabarró, J. (1988). *Structural Complexity*, volume 1. Springer.

Baru, C., Fecteau, G., Goyal, A., Hsiao, H., Jhingran, A., Padmanabhan, S., Copeland, G., and Wilson, W. (1995). DB2 Parallel Edition. *IBM Systems Journal*, 34(2):292–322.

Bayer, R. (1972). Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1(4):290–306.

Becker, L., Hinrichs, K., and Finke, U. (1993). A New Algorithm for Computing Joins with Grid Files. In *Proc. of the 9th International Conference on Data Engineering, Vienna, Austria*, pages 190–197.

Bellman, R. (1957). Dynamic Programming. Princeton University Press, Princeton, NJ, USA.

Bergsten, B., Couprie, M., and Valduriez, P. (1993). Overview of Parallel Architectures for Databases. *The Computer Journal*, 36(8).

Bhide, A. and Stonebraker, M. (1988). A Performance Comparison of Two Architectures for Fast Transaction Processing. In *Proc. of the 4th International Conference on Data Engineering, Los Angeles, CA, USA*, pages 536–545.

Blank, T. (1990). The MasPar MP-1 Architecture. Technical report, MasPar Computer Corporation.

Blasgen, M. and Eswaran, K. (1977). Storage and Access in Relational Databases. *IBM Systems Journal*, 16(4):363–377.

Bratbergsengen, K. (1984). Hashing Methods and Relational Algebra Operations. In *Proc. of the 9th Int. Conference on Very Large Data Bases (VLDB), Singapore*, pages 323–333.

Bronstein, I. and Semendjajew, K. (1987). *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 23rd edition.

Brooks, F. B. (1956). *The Analytic Design of Automatic Data Processing Systems*. PhD thesis, Harvard University, Cambridge, MA, USA.

Carino, F. and Kostamaa, P. (1992). Exegesis of DBS / 1012 and P-90 – industrial supercomputer database machine. In *Proc. of the 4th International PARLE Conference, Paris, France*, pages 877–892.

Cattell, R. (1996). *The Object Database Standard: ODMG-93 (release 1.2)*. Morgan Kaufmann.

Cekleov et al. (1993). SPARCceter 2000: Multiprocessing for the 90's! In *Proceedings COMPCON Spring'93, San Francisco*.

Chen, P. (1976). The Entity-Relationship Model: Toward A Unified View Of Data. *ACM Transactions on Database Systems*, 1(1):9–36.

Clark, N. (1997). Millenium Bug: Government accused of not doing enough. *Financial Times*. (20.3.97).

Clifford, J. and Croker, A. (1987). The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. In *Proc. of the 3rd Internat. Conference on Data Engineering, Los Angeles, USA*, pages 528–537.

Clifford, J., Dyreson, C., Isakowitz, T., Jensen, C., and Snodgrass, R. (1997). On the Semantics of "Now" in Databases. *ACM Transactions on Database Systems*, 22(2):171–214.

Clifford, J., Dyreson, C., Snodgrass, R., Isakowitz, T., and Jensen, C. (1994). Now in TSQL2. A TSQL2 Commentary.

Clifford, J. and Tuzhilin, A., editors (1995). *Recent Advances in Temporal Databases – Proc. of the International Workshop on Temporal Databases, Zürich, Switzerland*, Workshops in Computing. Springer.

CODASYL (1971). CODASYL Data Base Task Group April 71 Report. ACM, New York.

Codd, E. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6). Republished in *Milestones of Research – Selected Papers 1958-1982* (CACM 25th Anniversary Issue), CACM, 26(1), January 1983.

Compaq Computer Corp. (1997). Compaq Access – Configuration and Tuning of Microsoft SQL Server 6.5 for Windows NT on Compaq Servers. http://www.compaq.com / support / techpubs / whitepapers / 415a0696.html.

Conover, W. (1980). *Practical Nonparametric Statistics*. John Wiley & Sons, New York, 2nd edition.

Cray Research (1993). *Cray T3D System Architecture Overview*. Cray Research Inc. HR-04033 edition.

Darwen, H. (1997). Developments in SQL3. personal conversation.

Date, C. (1995). *An Introduction to Database Systems*, volume I. Addison-Wesley, 6th edition.

Davies, C., Lazell, B., Hughes, M., and Cooper, L. (1995). Time is just another attribute – or at least, just another dimension. In [Clifford and Tuzhilin, 1995], pages 175–193.

DeWitt, D. and Gerber, R. (1985). Multiprocessor hash-based join algorithms. In *Proc. of the 11th Intern. Conf. on Very Large Data Bases (VLDB), Stockholm, Sweden*, pages 151–164.

DeWitt, D. and Gray, J. (1990). Parallel Database Systems: The Future of Database Processing or a Passing Fad. *ACM SIGMOD RECORD*, 19(4):104–112.

DeWitt, D. and Gray, J. (1992). Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98.

DeWitt, D., Naughton, J., and Schneider, D. (1991). An Evaluation of Non-Equijoin Algorithms. In *Proc. of the 17th International Conference on Very Large Data Bases, Barcelona, Spain*, pages 443–452.

DeWitt, D., Naughton, J., Schneider, D., and Seshradi, S. (1992). Practical Skew Handling in Parallel Joins. In *Proc. of the 18th International Conference on Very Large Data Bases, Vancouver, Canada*, pages 27–40.

El-Masri, R. and Navathe, S. (1994). *Fundamentals of Database Systems*. Addison-Wesley, 2nd edition.

Elmasri, R., Wuu, G., and Kouramajian, V. (1993). The Time Index and the Monotonic B^+-tree. In [Tansel et al., 1993], chapter 18, pages 433–456.

Gadia, S. (1988). A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Transactions on Database Systems*, 13(4):418–448.

Gadia, S. (1992). A Seamless Generic Extension of SQL for Querying Temporal Data. Technical Report TR-92-02, Computer Science Department, Iowa State University.

Garey, M. and Johnson, D. (1979). *Computer and Intractability – a Guide to the Theory of NP-Completeness*. Freeman.

Gerber, R. (1986). Dataflow Query Processing Using Multiprocessor Hash-Partitioned Algorithms. Technical Report 672, Computer Science Dept., University of Wisconsin.

Gibbons, P., Matias, Y., and Poosala, V. (1997). Fast Incremental Maintenance of Approximate Histograms. In *Proc. of the Intern. Conf. on Very Large Data Bases (VLDB), Athens, Greece*, pages 466–475.

Glass, R. (1997). The Next Date Crisis and the Ones After That. *Communications of the ACM*, 40(1):15–17.

Gong, Y., Chuan, C., and Xiaoyi, G. (1996). Image Indexing and Retrieval Based on Color Histograms. *Multimedia Tools and Applications*, 2(2):133–156.

Goyal, P., Li, H., Regener, E., and Sadri, F. (1988). Scheduling of Page Fetches in Join Operations Using Bc-Trees. In *Proc. of the 4th Int. Conference on Data Engineering, Los Angeles, USA*, pages 304–310.

Graefe, G. (1993). Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170.

Graefe, G. (1994). Sort-Merge-Join: An Idea Whose Time Has(h) Passed? In *Proc. of the 10th Int. Conference on Data Engineering, Houston, USA*, pages 406–417.

Gray, J. (1995). A Survey of Parallel Database Techniques and Systems. Tutorial given at the 21st International Conference on Very Large Data Bases (VLDB), Zürich, Switzerland.

Gunadhi, H. and Segev, A. (1990). A Framework for Query Optimization in Temporal Databases. In Michalewicz, Z., editor, *Proc. of the 5th International Conf. on Statistical and Scientific Database Management, Charlotte, NC, USA*, number 420 in Lecture Notes in Computer Science (LNCS), pages 131–147. Springer.

Gunadhi, H. and Segev, A. (1991). Query Processing Algorithms for Temporal Intersection Joins. In *Proc. of the 7th International Conference on Data Engineering, Kobe, Japan*, pages 336–344.

Gunadhi, H. and Segev, A. (1993). Efficient Indexing Methods for Temporal Relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):496–509.

Günther, O. (1993). Efficient Computation of Spatial Joins. In *Proc. of the 9th International Conference on Data Engineering, Vienna, Austria*, pages 50–59.

Heinrich, C. and Hofmann, M. (1996). Decision Support from the SAP Open Information Warehouse. White paper, SAP AG, Walldorf, Germany. available via http://www.sap-ag.de/r3/pdf/oiw_e.pdf.

Hillis, D. (1985). *The Connection Machine*. MIT Press, Cambridge, MA, USA.

Hinrichs, K. and Nievergelt, J. (1983). The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects. In *Proc. of the 1983 Workshop on Graphtheoretic Concepts in Computer Science*, pages 100–113.

Hua, K. and Lee, C. (1991). Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. In *Proc. of the 17th International Conference on Very Large Databases (VLDB), Barcelona, Spain*.

Hua, K., Lee, C., and Peir, J.-K. (1991). Interconnecting Shared-Everything Systems for Efficient Parallel Query Processing. In *Proceedings of the 1st International Conference on Parallel Distributed Information Systems, Miami Beach, FL, USA*, pages 262–270.

Hyafil, L. and Rivest, R. (1973). Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems. Rapport de Recherche 33, IRIA-Laboria, Domaine de Voluceau, Rocquencourt, 78150 Le Chesnay, France.

Informix Inc. (1995). Informix & Data Warehousing. http://www.informix.com/informix/solution/warehous/intro.htm.

Inmon, W. (1996). *Building the Data Warehouse.* John Wiley & Sons, 2nd edition.

International Data Corporation (IDC) (1996). The Financial Impact of Data Warehousing. extract available via http:// www.sagus.com/ prod-info/ dw/ idcexec.htm.

Ioannidis, Y. and Christodoulakis, S. (1993). Optimial Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results. *ACM Transactions on Database Systems*, 18(4):709–748.

ISO92 (1992). International Organization for Standardization / International Electrotechnical Commission – Database Language SQL. ISO / IEC 9075.

Jensen, C., Clifford, J., Elmasri, R., Gadia, S., Hayes, P., and Jajodia, S. (1994a). A Consensus Glossary of Temporal Database Concepts. *SIGMOD Record*, 23(1):52–64.

Jensen, C., Clifford, J., Gadia, S., Segev, A., and Snodgrass, R. (1992). A Glossary of Temporal Database Concepts. *SIGMOD Record*, 21(3).

Jensen, C., Mark, L., and Roussopoulos, N. (1991). Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):461–473.

Jensen, C., Snodgrass, R., and Soo, M. (1994b). The TSQL2 Data Model. A TSQL2 Commentary.

Jensen, C., Soo, M., and Snodgrass, R. (1994c). Unifying Temporal Data Models via a Conceptual Model. *Information Systems*, 19(7):513–547.

Keller, A. and Roy, S. (1991). Adaptive Parallel Hash Join in Main-Memory Databases. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 58–67.

Kernighan, B. (1971). Optimal Sequential Partitions of Graphs. *Journal of the ACM*, 18(1):34–40.

Kim, W. (1980). A New Way to Compute the Product and Join of Relations. In *Proc. of the ACM SIGMOD Int. Conference on Management of Data, Santa Monica, USA*, pages 179–187.

Kitsuregawa, M., Harada, L., and Takagi, M. (1989). Join Strategies on Kd-Tree Indexed Relations. In *Proc. of the 5th Int. Conference on Data Engineering, Los Angeles, USA*, pages 85–93.

Kitsuregawa, M. and Ogawa, Y. (1990). Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC). In McLeoad, D., Sacks-Davis, R., and Schek, H., editors, *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB), Brisbane, Australia*, pages 210–221. Morgan Kaufmann.

Kitsuregawa, M., Tanaka, H., and Moto-Oka, T. (1983). Application of Hash to Database Machine and its Architecture. *New Generation Computing*, 1(1).

Kitsuregawa, M., Tanaka, H., and Moto-oka, T. (1984). Architecture and Performance of Relational Algebra Machine Grace. In *Proceedings of the International Conference on Parallel Processing, Chicago, Illinois, USA*.

Kline, N. (1993). An Update of the Temporal Database Bibliography. *SIGMOD Record*, 22(4):66–80.

Knuth, D. (1973). *The Art of Computer Programming – Sorting and Searching*, volume 3. Addison-Wesley.

Kolovson, C. (1993). Indexing Techniques for Historical Databases. In [Tansel et al., 1993], chapter 17, pages 418–432.

Korth, H. and Silberschatz, A. (1991). *Database System Concepts*. McGraw-Hill, 2nd edition.

Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. (1991). The ObjectStore System. *Communications of the ACM*, 34(10):50–63.

Lehman, T. and Carey, M. (1986). Query Processing in Main Memory Database Management Systems. In *Proc. of the ACM SIGMOD Int. Conference on Management of Data, Washington, USA*, pages 239–250.

Leung, T. and Muntz, R. (1990). Query Processing for Temporal Databases. In *Proc. of the 6th International Conference on Data Engineering, Los Angeles, CA, USA*, pages 200–208.

Leung, T. and Muntz, R. (1992). Temporal Query Processing and Optimization in Multiprocessor Database Machines. In *Proc. of the 18th International Conference on Very Large Data Bases, Vancouver, Canada*, pages 383–394.

Leung, T. and Muntz, R. (1993). Stream Processing: Temporal Query Processing and Optimization. In [Tansel et al., 1993], chapter 14, pages 329–355.

Lo, M.-L. and Ravishankar, C. (1996). Spatial Hash-Joins. In *Proceedings ACM SIGMOD Conference on Management of Data, Montreal, Canada*, pages 247–258.

Lockemann, P., Krüger, G., and Krumm, H. (1993). *Telekommunikation und Datenhaltung*. Studienbücher der Informatik. Hanser Verlag.

Lorentzos, N. and Mitsopoulos, Y. (1994). SQL Extension for Interval Data. Technical Report 105, Informatics Laboratory, Agricultural University of Athens.

Lorentzos, N. and Mitsopoulos, Y. (1997). SQL Extension for Interval Data. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):480–499.

Lu, H., Ooi, B.-C., and Tan, K.-L. (1994). On Spatially Partitioned Temporal Join. In *Proc. of the 20th Internat. Conf. on Very Large Data Bases (VLDB), Santiago de Chile*, pages 546–557.

Mannino, M., Chu, P., and Sager, T. (1988). Statistical Profile Estimation in Database Systems. *ACM Computing Surveys*, 20(3):191–221.

Mishra, P. and Eich, M. (1992). Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113.

Ng, R. and Tam, D. (1997). Analysis of Multilevel Color Histograms. In *Proc. of the Conf. on Storage and Retrieval for Image and Video Databases, San Jose, CA, USA*, pages 22–33.

Norman, M. and Thanisch, P. (1995). *Parallel Database Technology: An Evaluation and Comparison of Scalable Systems*. The Bloor Research Group, UK. ISBN 1-874160-17-1.

Norman, M., Zurek, T., and Thanisch, P. (1996). Much Ado about Shared-Nothing. *SIGMOD Record*, 25(3):16–21.

O'Neil, P. and Graefe, G. (1995). Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record*, 24(3):8–11.

Oracle Corp. (1996). Data Warehousing – Informed Decision Making. http://www.oracle.com/initiatives/wti/html/index.html.

Orenstein, J. (1986). Spatial Query Processing in an Object-Oriented Database System. In *Proc. of the ACM SIGMOD Int. Conference on Management of Data, Washington, USA*, pages 326–336.

Orenstein, J. and Manola, F. (1988). PROBE Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Transactions on Software Engineering*, 14(5):611–629.

Patel, J. and DeWitt, D. (1996). Partition Based Spatial-Merge Join. In *Proceedings ACM SIGMOD Conference on Management of Data, Montreal, Canada*, pages 259–270.

Perrizo, W., Gustafson, J., Thureen, D., and Wenberg, D. (1991). Domain Vector Accelerator (DVA): A Query Accelerator for Relational Operations. In *Proc. of the 7th International Conference on Data Engineering, Kobe, Japan*, pages 491–498.

Piatetsky-Shapiro, G. and Connell, C. (1984). Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Proceedings ACM SIGMOD 1984 Conference on Management of Data*, pages 256–276.

Pissinou, N., Snodgrass, R., Elmasri, R., Mumick, I., Tamer Özsu, M., Pernici, B., Segev, A., Theodoulis, B., and Dayal, U. (1994). Towards an Infrastructure for Temporal Databases – Report of an Invitational ARPA / NSF Workshop (June 1993). Technical Report TR 94-01, Dept. of Computer Science, University of Arizona.

Poosala, V., Ioannidis, Y., Haas, P., and Shekita, E. (1996). Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings ACM SIGMOD Conference on Management of Data, Montreal, Canada*, pages 294–305.

Prism Solutions Inc. (1996). Why a Data Warehouse. http://www.prismsolutions.com/data/why.html.

Rana, S. and Fotouhi, F. (1993). Efficient Processing of Time-Joins in Temporal Data Bases. In *Proc. of the 3rd Internat. Symposium on Database Systems for Advanced Applications*, pages 427–432.

Red Brick Systems (1995a). Specialized Requirements for Relational Data Warehouse Servers. White paper, Red Brick Systems.

Red Brick Systems (1995b). Star Schemas and STARjoin Technology. White paper, Red Brick Systems. available via *http://www.redbrick.com/rbs/whitepapers/star_wp.html*.

Red Brick Systems (1995c). The Data Warehouse. White paper, Red Brick Systems.

Ryan, N. and Smith, D. (1995). *Database Systems Engineering*. Thomson, 1st edition.

Sarda, N. (1993). HSQL: A Historical Query Language. In [Tansel et al., 1993], chapter 5, pages 110–140.

Schneider, D. and DeWitt, D. (1989). A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. Technical Report TR-836, Computer Science Dept., University of Wisconsin.

Seagate Technology (1997). Ultra SCSI – Technical Fact Sheet. http://www.seagate.com/support/disc/papers/ultrafs.shtml.

Segev, A. (1993). Join Processing and Optimization in Temporal Relational Databases. In [Tansel et al., 1993], chapter 15, pages 356–387.

Segev, A. and Gunadhi, H. (1989). Event-Join Optimization in Temporal Relational Databases. In *Proc. of 15th Internat. Conf. on Very Large Data Bases (VLDB), Amsterdam, Netherlands*, pages 205–215.

Segev, A., Gunadhi, H., Chandra, R., and Shanthikumar, J. (1993). Selectivity Estimation of Temporal Data Manipulations. *Information Sciences*, 74(1/2):111–149.

Shapiro, L. (1986). Join Processing in Database Systems with Large Memories. *ACM Transactions on Database Systems*, 11(3):239–264.

Sloan, R. (1992). A Practical Implementation of the Data Base Machine – Teradata DBC/1012. In *Proc. of the 25th Hawaii International Conference on System Sciences, Kauai, HI, USA*, pages 320–327.

Snodgrass, R. (1987). The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298.

Snodgrass, R., editor (1995). *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers.

Snodgrass, R. (1996). A Road Map of Additions to SQL/Temporal. Change Proposal, available via ftp from `ftp.cs.arizona.edu`.

Snodgrass, R., Ahn, I., Ariav, G., Batory, D., Clifford, J., Dyreson, C., Elmasri, R., Grandi, F., Jensen, C., Käfer, W., Kline, N., Kulkarni, K., Leung, T., Lorentzos, N., Roddick, J., Segev, A., Soo, M., and Sripada, S. (1994). TSQL2 Language Specification. *SIGMOD RECORD*, 23(1):65–86.

Soo, M., Snodgrass, R., and Jensen, C. (1994). Efficient Evaluation of the Valid-Time Natural Join. In *Proc. of the 10th International Conference on Data Engineering, Houston, Texas, USA*, pages 282–292.

Stonebraker, M. (1986). The Case for Shared Nothing. *IEEE Data Engineering*, 9(1).

Stonebraker, M. (1987). The Postgres Data Model. In *Proc. of the 13th International Conference on Very Large Data Bases (VLDB), Brighton, England*, pages 83–96.

Stonebraker, M., Rowe, L., and Hirohama, M. (1990). The Implementation of Postgres. *IEEE Transaction on Knowledge and Data Engineering*, 2(1):125–142.

Su, S. (1988). *Database Computers: Principles, Architectures, and Techniques*. McGraw-Hill.

Tandem Computers GmbH (1997). System-Interconnect-Technologie. http://www.tandem.de/technik/svrnet.htm.

Tannenbaum, A. (1994). *Distributed Operating Systems*. Prentice-Hall.

Tansel, A. (1986). Adding Time Dimension to Relational Model and Extending Relational Algebra. *Information Systems*, 11(4):343–355.

Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., and Snodgrass, R. (1993). *Temporal Databases – Theory, Design and Implementation*. Benjamin/Cummings.

Teradata Corporation (1983). DBC/1012 Data Base Computer Concepts & Facilities. Technical Report Document No. C02-0001-00, Teradata Corporation.

Teradata Corporation (1985). *DBC/1012 Database Computer System Manual Release 2.0*. Document No. C10-0001-02.

Tseng, E. and Reiner, D. (1993). Parallel Database Processing on the KSR1 Computer. In *Proc. of the 1993 ACM SIGMOD Conference, Washington DC, USA*, page 453ff.

Tsichritzis, D. and Lochovsky, F. (1977). *Data Base Management Systems*. Academic Press, New York.

Tsotras, V. and Kumar, A. (1996). Temporal Database Bibliography Update. *SIGMOD Record*, 25(1).

Uhlig, R. (1997). Millenium Computer Chaos 'will cost £ 31 bn'. *Electronic Telegraph*. (12.4.97).

Valduriez, P. (1987). Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246.

Valduriez, P. (1993a). Parallel Database Systems: open problems and new issues. *Distributed and Parallel Databases*, 1(2):137–165.

Valduriez, P. (1993b). Parallel Database Systems: the case for shared-something. In *Proc. of the 9th International Conference on Data Engineering, Vienna, Austria*, pages 460–465.

Walton, C., Dale, A., and Jenevein, R. (1991). A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In Lohman, G. M., Sernadas, A., and Camps, R., editors, *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 537–548. Morgan Kaufman San Mateo.

Wang, X. and Luk, W. (1988). Parallel Join Algorithms on a Network of Workstations. In *Proc. of Intern. Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, USA*, pages 87–96.

Witkowski, A. e. a. (1993). NCR 3700 – the next-generation industrial database computer. In *Proceedings of the 19th International Conference on Very Large Data Bases, Dublin, Ireland*, pages 230–243.

Wolf, J., Dias, D., and Yu, P. (1990). An Effective Algorithm for Parallelizing Sort-Merge Joins in the Presence of Data Skew. In *Proc. of the 2nd International Symposium on Parallel and Distributed Systems, Dublin, Ireland*.

Wolf, J., Dias, D., and Yu, P. (1993). A Parallel Sort Merge Join Algorithm for Managing Data Skew. *IEEE Transactions on Parallel and Distributed Processing*, 4(1):70–86.

Zhou, H. (1993). Two-stage m-way graph partitioning. *Parallel Computing*, 19(12):1359–1373.

Zipf, G. (1949). *Human Behaviour and the Principle of Least Effort*. Addison-Wesley.

Zurek, T. (1996). Parallel Temporal Nested-Loop Joins. Technical Report ECS-CSG-20-96, Dept. of Computer Science, Edinburgh University.

Zurek, T. (1997a). Optimal Interval Partitioning for Temporal Databases. In *Proc. of the 3rd BIWIT Workshop, Biarritz, France*, pages 140–147. IEEE Computer Society Press.

Zurek, T. (1997b). Optimisation of Partitioned Temporal Joins. In *Proc. of the 15th BNCOD Conference, London, UK, LNCS 1271*, pages 101–115. Springer.

Zurek, T. (1997c). Parallel Processing of Temporal Joins. To appear in *Informatica*, ISSN 0350-5596.

Zurek, T. (1997d). Parallel Temporal Joins. In *"Datenbanksysteme in Büro, Technik und Wissenschaft" (BTW), German Database Conference, Ulm, Germany*, pages 269–278. Springer. (in English).

Zurek, T., Minty, E., and Thanisch, P. (1996). Recursive Query Processing on a Connection Machine. In Keane, J., editor, *Parallel Information Processing*, pages 211–237. UNICOM, Stanley Thornes Publishers.

Zurek, T. and Thanisch, P. (1995). Strategies for Parallel Linear Recursive Query Processing. In Sellis, T., editor, *Proc. of the 2nd International Workshop "Rules in Database Systems" (RIDS), Glyfada, Athens, Greece*, LNCS 985, pages 213–229. Springer.

Index