Data Partitioning in Parallel Temporal Databases

A PhD Thesis Proposal

Thomas Zurek

September 4, 1995

Abstract

Temporal relational query processing can impose considerable demands on the performance of a database management system. The main reasons are the huge size of temporal relations and the complexity of temporal operators. Parallelism can offer solutions to achieve the necessary performance.

In order to successfully implement parallel algorithms for temporal query operators one needs to partition the data over the temporal attribute(s). Latter are usually represented by intervals or rectangles. This makes the data partitioning problem much more complex because data has to be replicated in most cases in order to guarantee the correctness of the respective operation. This can cause significant overhead costs. Such an effect does not exist in conventional parallel database systems.

This paper motivates and proposes a PhD research project for finding and designing techniques to partition temporal relations over their temporal attribute(s). These techniques should minimise the overhead costs caused by the replication of tuples. Only expensive algorithms can find optimal solutions to this problem if they can be found at all (this depends on the actual cost function). Therefore it is necessary to find fast strategies that offer near-optimal solutions. I believe that these can be based on heuristics using statistical information obtained from the database catalog and data sampling.

The paper concludes with a description of the stages and methods of the project. It also gives a rudimentary timetable for performing the stages.

Contents

1 Introduction and Overview

In this research project I want to combine the areas of parallel and temporal databases, both restricted to the relational database model[1] (Codd, 1970). This combination is very promising: parallel database technology can solve the performance problems imposed by temporal databases. This is essential in order to successfully incorporate temporal database features into future database products.

The two areas have reached different stages of development:

- Temporal databases are a relatively new but increasingly busy area of research. This can be seen from growing number of papers on temporal database issues that have been published in recent years (Kline, 1993). More and more presentations on temporal database research issues are given on the main worldwide database conferences. Recent efforts have evolved in a standard for a temporal (relational) query language, namely TSQL2 (Snodgrass et al., 1994a). Future targets[2] are the incorporation of temporal features in the new SQL3 standard which is currently under discussion and expected for 1996.
- Parallel database technology, on the other hand, has reached a high standard and already found its way into commercial products (DeWitt and Gray, 1992). Parallelism has managed to overcome performance deficits of traditional mainframe databases. Each of the big database vendors like Oracle, Sybase, Informix etc. offer products based on parallel technology. Furthermore each of the big vendors of parallel hardware like Cray, IBM, Sequent, Convex, Pyramid etc. cooperates with several of the database vendors as databases are regarded as the dominant part of the parallel market in the future (Agerwala et al., 1995).

Relational query language like SQL are *declarative*, i.e. they describe the result rather than the procedure to compute the result. In particular, queries are not explicitly parallel. This releases a user from the problem of formulating the query in the most efficient way and makes this problem the responsibility of the database management system. There, the declarative query is translated into a procedural expression which is then optimised. One optimisation problem is to choose the most efficient algorithm for certain operations. Database management systems, for example, usually have around a dozen options of how to compute a join, depending on the sizes of the relations involved or the join condition (e.g. equality or non-equality predicates).

Temporal database impose a variety of problems to this process:

- The query optimiser has to exploit the temporal semantics. Temporal join conditions, for example, translate to non-equality expressions. This differs from the usual case that is dominated by the use of equijoins. Usually, the database management systems provides join algorithms that make use of such equality conditions. For the very rare non-euality-cases all-purpose algorithms are used which are much slower. Traditional database systems can therefore not process temporal joins efficiently.

[1]The remaining part of the proposal assumes that the reader is familiar with the notion and terminology of relational database systems. Introductions to these topics can be found in many textbooks that discuss databases like (Korth and Silberschatz, 1991), (Date, 1995), (Lockemann and Schmidt, 1987), (Lockemann et al., 1993) and many others.

[2]e.g. formulated in (Pissinou et al., 1994) and to be discussed on the "Workshop on Temporal Databases", 17.-18. September 1995, Zürich, Switzerland.

- There are further very demanding performance requirements, mainly created by
  - the – in comparison to non-temporal relations – immensely increased size of temporal relations due to the fact that tuples are *logically* deleted rather than *physically* removed, and
  - the additional computational complexity imposed by temporal operators (see e.g. (Soo et al., 1994a), (Tuzhilin and Clifford, 1990)).

As previously stated, parallel technology has already and successfully helped to bridge performance gaps in traditional database technology. Basically there are two properties for its scientific *and* commercial success:

- Performance of parallel database systems is scalable by adding new processing nodes to the underlying hardware – a relatively cheap and easy process.
- The parallel paradigm offers various possibilities to cut down computationally expensive tasks.

This suggests that successfully incorporating parallelism in temporal query processing can be beneficial not only from the scientific but also from the commercial point of view.

In section 2, I will give an overview of the current state of parallel database research and technology. Data partitioning will be described as the basic element to achieve parallelism in relational databases. Furthermore, certain characteristics of parallel database research and market developments of the past are looked at. They are important to be taken into consideration in the future in order to avoid errors that were committed in the past.

Section 3 introduces temporal databases in some more detail. I will point to the features that are essential with respect to parallel temporal query processing. Whereas data partitioning is relatively easy and straightforward for traditional relational databases it becomes more complex in the case of temporal data. Temporal interval and rectangle partitioning is a harder problem because of the overhead caused by the forced replication of data – a fact that does not exist in the traditional case.

Section 4 describes the work I have done so far. It motivates the PhD research project that I propose in section 5.

Section 6 states in some more detail the stages of the project. It gives research targets and concludes with a timetable for completing the project.

2 Parallel Databases – State of the Art

2.1 Overview

This section will introduce the area of parallel databases. There are several ways in which databases can be made to work in parallel. Section 2.2 will show the possible approaches. Amongst these approaches, the intra-operator parallelism is potentially the most beneficial for reasons explained below.

Another main characteristic of parallel relational databases is the fact that the relations / tables are horizontally partitioned with each fragment stored on another disk. This allows to increase the I/O bandwidth which is usually the main bottleneck of relational database systems. Data partitioning can also exploited by the algorithms implementing the relational algebra operators as operators can be applied to several data fragments in parallel in order to compute the result.

In fact, data partitioning is widely regarded as the main characteristic of parallel databases (Valduriez, 1993b) (DeWitt and Gray, 1992). It is a crucial step as it determines the effectiveness of the parallel execution. Section 2.3 therefore describes the ideas, prerequisites and problems of data partitioning in some more detail. This is necessary in order to understand the problems that arise with respect to parallel temporal databases (see section 3.3) which are the problems that my PhD thesis will focus on.

Finally, section 2.4 will give a sketch of the discussion in industry and the international research community around the question what parallel architectures should be considered for parallel databases. There is no consensus on this up to now although some conclusions can already be drawn from this discussion.

2.2 Forms of Parallelism in Databases

Essentially there are two granularities in database processing that can be parallelised:

- the processing of the queries that are present in the processing space at the same time,
- the processing of one single query

These determine the two main categories 1. and 2. of forms of parallelism that can be exploited by a database management system:

1. **inter-query parallelism**, i.e. between several queries running concurrently in parallel,
2. **intra-query parallelism**, i.e. inside a single query. A query can be represented as an algebraic expression consisting itself of several subexpressions linked by operators. Based on this view three subcategories can be identified:
   (a) **pipelining:** output of one operator is streamed into the input of another operator such that the computations of the two operators proceed in parallel like two cog wheels. A query $\pi(\sigma(R))$ for instance can be processed by streaming the resulting tuples of the selection σ as soon as they emerge into the projection π such that both operations proceed in parallel. There is a strong correspondence to stream processing approaches (Parker et al., 1989), (Leung and Muntz, 1993).

(b) **inter-operator parallelism:** two or more subexpressions are evaluated in parallel. A query $A \bowtie B \bowtie C \bowtie D$ can be processed by first evaluating the subexpressions $A \bowtie B$ and $C \bowtie D$ in parallel;

(c) **intra-operator parallelism:** parallel algorithms are implemented for the (relational algebra) operators. The common technique is to partition the data into several fragments – one per processor – and to perform locally, on each processor, the respective operation. Relational operators suit very well for this due to their algebraic properties (Zurek, 1995).

In inter-query parallelism queries might interfere because they access the same data. The main problem is therefore to manage the cooperation over a common data space. This is not an entirely database-specific problem but one common to many areas where parallelism applies. Nevertheless, the specific characteristics and prerequisites can be taken into account in each particular case. Generally, this problem is solved by designing protocols to guarantee the correctness of the cooperation. This approach is quite well understood and a lot of commercial parallel database products exploit this type of parallelism.

The nature of intra-query parallelism is quite different. It applies to the operators and the sequence of operators that represent the query. These operators naturally interfere not only on a common data space (as in the previous case) but also depend on algebraic constraints amongst each other, like commutativity or associativity. Finally, the parallel approach can step down to an algorithmic level (intra-operator parallelism).

Among the three forms of intra-query parallelism the intra-operator type seems to be the most beneficial. Relational queries (and their corresponding relational algebra expressions) do not tend to have long pipelines (DeWitt and Gray, 1992) and complex tree structures. Performance benefits through pipelining can therefore be expected to be very low. The same accounts for inter-operator parallelism because there are not a lot of possibilities to cut the task of processing the query down into a large number of subtasks that could be processed in parallel.

On the other hand, intra-operator parallelism based on data partitioning offers much better opportunities. It allows to parallelise expensive operators like the join. Although data partitioning is not specific to intra-operator parallelism[3] it is here where it can be exploited most. The reasons for this are explained in the following section.

2.3 Data Partitioning in Relational Databases

This section will introduce the notion of data partitioning for relational databases in some more detail. The relational data model offers a variety of possibilities to naturally exploit data partitioning: its algebra has a small set of uniform and simple operators working on large sets of data. Most operators are able to process an arbitrary portion of the input data to produce the portion of the result related to the input. For example selecting all tuples of relation R holding value x in attribute a can be broken up into several selections:

$$\begin{aligned} \sigma_{a=x}(R) &= \sigma_{a=x}(R_1 \cup \cdots \cup R_m) \\ &= \sigma_{a=x}(R_1) \cup \cdots \cup \sigma_{a=x}(R_m) \end{aligned} \tag{1}$$

[3] Valduriez even defines a parallel database system as a DBMS "which is implemented on a tightly-coupled (rem.: to distinguish it from a distributed database system) multiprocessor and exploits data partitioning" (Valduriez, 1993b).

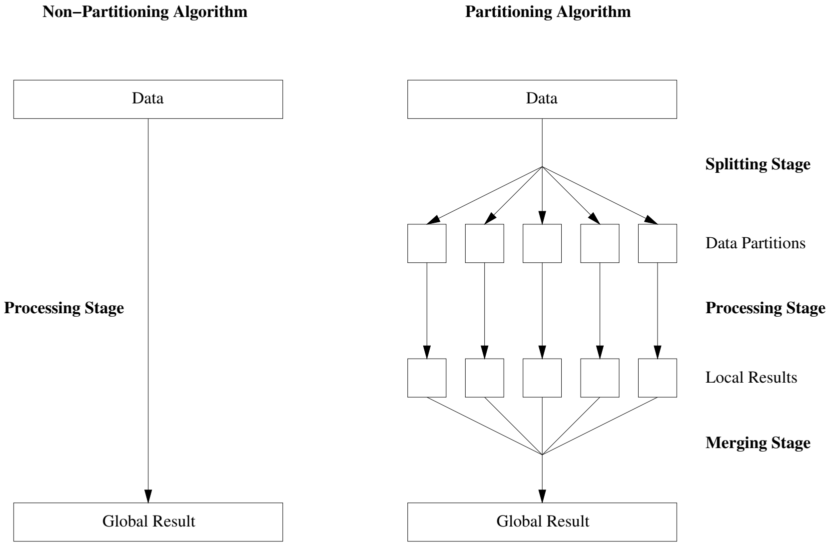

Figure 1: Stages of a non-partitioning and a partitioning algorithm.

This is due to the distributive property of the union operator $\cup$ with all of the other essential relational algebra operators (Korth and Silberschatz, 1991), i.e. selection σ, projection π, rename ρ, set difference $-$ and cartesian product $\times$. Additional operators like intersect $\cap$, all sorts of joins $\bowtie$, division $\div$ etc. are defined upon these kernel operators and have also this property as σ in (1). This allows us to easily split the data into several portions or – in other words – a single, probably big data flow can be split into several small ones.

Section 2.3.1 describes the structure of the algorithms that exploit this property. Section 2.3.2 shows that partitioning gets more complex for binary operators. Finally, data partitioning is relevant on several levels of memory, in particular main memory and secondary storage; section 2.3.3 discusses this issue.

2.3.1 Partitioning Algorithms

Algorithms based on partitioning essentially consist of three stages:

(a) The data is divided into a certain number of portions; this is called the *splitting stage*.

(b) An operation is executed on each portion of data; this is called the *processing stage*.

(c) The results of the processing stage are merged to build the global result; this is called the *merging stage*.

If we resume (1) then performing a single selection on R (left hand side of (1)) corresponds to performing a non-partitioning algorithm. If the cost function of a selection is $c(n)$ where n is e.g. the number of tuples in R then the total costs c_1 for the selection is

$$c_1 := c(n)$$

The right hand side of (1) corresponds to performing a partitioning algorithm. If we assume that R_i holds $n_i \approx n/m$ tuples[4] (for $1 \leq i \leq m$), splitting costs are $sc(n)$ and merging costs $mc(n)$ then the total costs of the partitioning algorithm are

$$c_2 := sc(n) + max_{i=1}^{m}(c(n_i)) + mc(n) \;\approx\; sc(n) + c(n/m) + mc(n) \tag{2}$$

The benefits are obvious: $c_2 \ll c_1$ when partitioning and merging (which is e.g. performing a union of the local results) are cheap and c is not a constant function, i.e. $c \notin O(1)$.

2.3.2 Partitioning for Binary Operators

In the previous sections the example of partitioning a selection was used to show the benefits of partitioning algorithms. The idea was to split the input relation into small portions and to take advantage of property (1). This works fine when the operator – in that case σ – is unary. In the case of a binary operator, like a join over a certain condition C, property (1) still holds:

$$\begin{aligned} R \bowtie_C S &= R \bowtie_C (S_1 \cup \cdots \cup S_m) \\ &= \sigma_C(R \times (S_1 \cup \cdots \cup S_m)) \\ &= \sigma_C((R \times S_1) \cup \cdots \cup (R \times S_m)) \\ &= \sigma_C(R \times S_1) \;\cup \cdots \cup\; \sigma_C(R \times S_m) \\ &= (R \bowtie_C S_1) \;\cup \cdots \cup\; (R \bowtie_C S_m) \end{aligned}$$

The problem is that only relation S is split; R has either to be replicated or to be accessed m times. In either way an overhead is produced due to additional communication (R is replicated) or disk access cost (multiple access). Therefore it is desirable to achieve a partition of R *and* S without loosing parts of the result and avoiding as much overhead as possible. Such a partition $R_1, \ldots, R_m$ and $S_1, \ldots, S_m$ of R and S respectively holds

$$R \bowtie_C S = (R_1 \bowtie_C S_1) \;\cup \cdots \cup\; (R_m \bowtie_C S_m) \tag{3}$$

and – ideally – the sets R_i (S_i respectively) are pairwise disjoint.

Finding such a partition is not always straightforward. In the case of an equi-join, i.e. a join over one or more equality predicates, a common technique is the following:

Let us assume that the join condition holds a predicate $R.a = S.b$ which has to hold for every tuple qualifying for the result. Let us further assume that the set of values occurring in the attributes $R.a$ and $S.b$ is $\{x_1, \ldots, x_m\}$. Then a partition for R and S that satisfies (3) and that produces pairwise disjoint sets R_i (S_i respectively) is

$$\begin{aligned} R_i &:= \{r \in R : r.a = x_i\} \\ S_i &:= \{s \in S : s.b = x_i\} \end{aligned} \tag{4}$$

for $1 \leq i \leq m$. (4) means that tuples $r \in R$ and $s \in S$ are in partition i if they hold value x_i in the respective join attribute. As stated above, the condition for the join involves $R.a = S.b$.

[4]The approximation represents the ideal, well balanced case. Problems of unbalanced partitions will be discussed later.

Thus tuples of R_i and S_j with $j \neq i$ do not need to be checked for the join condition. This implies that the number of computational steps has been reduced at the same time: if $|R|, |S|, |R_i|, |S_i|$ denote the number of tuples in the various sets then the original worst case complexity of the join $R \bowtie S$ is $|R| \cdot |S|$. Assuming that the values x_i are distributed uniformly, i.e.

$$|R_i| \approx \frac{|R|}{m}, \quad |S_i| \approx \frac{|S|}{m} \tag{5}$$

for $1 \leq i \leq m$. Then the number of steps imposed by (4) is

$$|R_1| \cdot |S_1| \; + \; \cdots \; + \; |R_m| \cdot |S_m| \; \approx \; m \cdot \frac{|R|}{m} \cdot \frac{|S|}{m} \; = \; \frac{|R| \cdot |S|}{m} \; \leq \; |R| \cdot |S| \tag{6}$$

In other words: the benefits of partitioning in this case lie not only in parallel processing but also in reducing the number of computational steps, which makes partitioning also a beneficial technique for sequential algorithm design, e.g. in (Soo et al., 1994b).

A problem arises when the data is not uniformly distributed as assumed in (2), (5) and (6). The effect of a non-uniform distribution of the data is called *data skew*. In the presence of data skew the approximations made in (2), (5) and (6) do not hold anymore; c_2 in (2) might still be less than c_1 in most cases but the benefits decrease.

Over the last few years data skew has been widely recognised as a serious problem in parallel databases. The ability to cope with data skew has become an important criteria for parallel implementations of database operators, e.g. in (Zurek et al., 1995). A lot of research effort has been spent on data skew and on designing algorithms that have this ability. (Kitsuregawa and Ogawa, 1990) (Walton and Dale, 1991) (Hua et al., 1991) (DeWitt et al., 1992) (Wolf et al., 1993) are examples of this development.

2.3.3 Levels of Data Partitioning

In parallel databases two levels of data partitioning can be identified:

1. As we have seen in section 2.3.1, the data is partitioned into several fragments. Each of these fragments is – at least partially – allocated to some part of the main memory and is accessed by one processor respectively, just before being processed. This means that data is partitioned on the main memory level.

2. In comparison to main memory, accesses to secondary storage are tremendously slow. The performance of database algorithms is therefore determined by the number and quality of accesses to secondary storage. In order to bridge this bottleneck and to increase the bandwidth, parallel databases usually hold their data partitioned over several disks to achieve a parallel I/O. In this case, data is partitioned the secondary storage level.

Ideally both levels should match each other, i.e. the partition should be the same. Usually this is not the case like in the following example:

Example: Two relations R and S, each holding an attribute *name*, are to be joined over the condition

$$R.name = S.name$$

Imagine that R is partitioned into 3 fragments, the first one holding all tuples with names starting with a letter in the range A–G, the second one covering range H–S and the third one T–Z. This might be a well-balanced partition for relation R. S might be partitioned differently, e.g. with ranges A–D, E–Q, R–Z. In order to compute a join $R \bowtie S$ we need a *single* partition for *both* relations[5]. This means that at least one of the partitions on secondary storage level has to be mapped to a different partition on main memory level.

Another view of this problem is that partitions on main memory level are dynamically created depending on the context, i.e. the query, whereas secondary storage partitions are static, i.e. context-independent, and essentially balance the loads between the fragments. In fact, latter problem is similar and naturally related to indices like B or B^+ trees that include mechanisms to maintain balance whenever nodes are inserted or removed (Bayer, 1972), (Knuth, 1973), (Nievergelt, 1974).

2.4 Parallel Database Architectures

An important issue around parallel algorithm design is the underlying parallel architecture. Two architectural levels can be considered:

1. the architecture of the parallel hardware (physical architecture),
2. the architecture imposed by the parallel software, the database management system (logical architecture).

There have been attempts to match the logical and the physical architectures in order to avoid the overhead caused by the mapping from one to the other. Economical considerations suggest that these special-purpose architectures might be too expensive, probably already for the relatively big market of non-temporal relational database but definitely for the small market of temporal databases. In long-term, one can expect parallel database systems to be running on all-purpose parallel architectures for reasons of economy although this might not be the optimal solution from the scientific point of view. All-purpose machines nowadays can be configured in various ways and be easily adapted towards the architecture imposed by the software. The following considerations therefore refer to logical architecture issues.

The underlying parallel architecture will determine the cost model on which the algorithm design takes place. But the question "which is the best architecture for parallel databases?" is still an open problem and is widely discussed in several papers (Stonebraker, 1986) (Hua et al., 1991) (DeWitt and Gray, 1992) (Valduriez, 1993b) (Valduriez, 1993a) (Bergsten et al., 1993) (Baru et al., 1995) and the discussion whether shared-memory, shared-disk or shared-nothing is best is not over yet.

In the late 1980s, several researchers pleaded for the shared-nothing, like (Stonebraker, 1986) and (DeWitt and Gray, 1992). Several academic and commercial systems based upon shared-nothing arose, e.g. Bubba (Boral et al., 1990), Gamma (DeWitt et al., 1990), Teradata DBC/1012 (Teradata Corporation, 1985), Tandem TXP (Englert et al., 1989), EDS (EDS Database Group, 1990), IBM Arbre (Lorie et al., 1989), Grace (Kitsuregawa et al., 1984). The arguments seemed to be evident: shared-nothing minimises interference by minimising resource sharing and they

[5] Note that this problem is again caused by binary operators; unary operators are uncritical.

scale up very well. In contrast, shared-memory (also refered to as *shared-everything*) and shared-disk systems seemed to be limited in their performance by the fact that some resources are shared. This causes interferences.

The main disadvantages of shared-nothing, namely high complexity and load balancing problems, obviously caused the industrial and scientific community to change the direction and to consider shared-memory and shared-disk approaches. Examples are XPRS (Stonebraker et al., 1988), DBS3 (Bergsten et al., 1991) or Volcano (Graefe, 1990).

Shared-memory benefits from its simplicity and the easy load balancing whereas shared-disk has advantages in extensibility (although not as far-reaching as in the shared-nothing approach), availability, load balancing and easy migration from uniprocessor systems.

Therefore several authors (Hua et al., 1991) (Valduriez, 1993b) (Bergsten et al., 1993) plead for a hybrid parallel architecture which Valduriez calls *shared-something*. It consists of two levels: a number of processing nodes sharing a pool of disks and each processing node is a shared-memory multiprocessor. In this way shared-something provides the advantages of shared-disk with the extensibility and cost of shared-nothing.

In summary: the parallel algorithm design is influenced by the underlying parallel architecture to a very high extent. There is, however, no consensus about the most adequate architecture for a parallel database system. Therefore it will be a major task of the PhD project to define an architecture that is precise enough for algorithm design but still general enough such that the results are generally applicable.

3 Temporal Databases – State of the Art

3.1 Overview

In this section, I will briefly describe the development of the research area in section 3.2, introduce the problem of data partitioning for temporal databases in section 3.3 and finally point to areas that are confronted with similar problems as those of parallel temporal databases (section 3.4).

3.2 Development of the Research Area

Over the last decades, time has emerged to be an important data entity. Many database applications hold time or time-varying information, e.g. holding patients medical records, i.e. the evolution of their medical state over time. Doctors usually analyse the states of illness of a patient and query over his record which might be stored in a medical database. The significant characteristic of this database is that not only the current state of an entity is kept in the database – in this example the patient's record – but also previous ones.

A lot of other applications show similar requirements. From the theoretical point of view, traditional, so called *snapshot*, relational database systems are able to cope with time and time-varying data without supporting a special temporal functionality. However, practical reasons, like user-friendly (and in particular orthogonal) query languages and acceptable performance, require facilities for

- defining and managing temporal relations,
- formulating and processing temporal queries.

In the 1970s and 1980s these problems were tackled by defining and implementing application-specific temporal data models. Such examples can be found in (Wiederhold et al., 1975) or (Jones et al., 1979). However, the demand grew to create application-independent models and systems as there is a large variety of database applications than involve temporal aspects.

Around two dozens of temporal models have been presented in the last 10 years. In 1994, a TSQL2 standard (Snodgrass et al., 1994a), an extension of the SQL92 standard (ISO92, 1992), was defined by a group of leading temporal database researchers. During this process common features of previously suggested temporal models were incorporated into a TSQL2 data model (Jensen et al., 1994a) (Jensen et al., 1994b). The standardisation process is now heading for TSQL3 and/or the incorporation of temporal features into the new SQL3 standard which is expected for 1996.

Currently, leading researchers mainly concentrate on temporal query language definition and modeling, probably in order to create a widely accepted basic framework for future developments in the field. Implementation issues of temporal databases, especially in connection with parallelism, have attracted relatively little attention. The only paper – to my knowledge – that discusses parallel issues of temporal databases is (Leung and Muntz, 1992); parts of which can be found in (Leung and Muntz, 1993). According to the authors, Leung and Muntz, they did not go on working in this area and do not know of any other researchers on related problems. However, there are research topics and areas that are related to that of parallelism for temporal databases. Section 3.4 is devoted to this aspect.

3.3 Data Partitioning in Temporal Databases

Data partitioning in temporal databases differs from the usual case (as described in section 2.3) only if

- the operation involves a time attribute[6] and
- data partitioning for the respective operator is sensitive to the semantics of the operator, i.e. the fact that not *any* partition is suitable (like round-robin partitioning for a join).

As a consequence, data partitioning for unary (temporal) operators like σ^T or π^T is uncritical and the same as in the non-temporal, the snapshot case. In the following I will therefore focus on partitioning for binary temporal operators, in particular the temporal join.

The temporal join is the most expensive temporal database operator. It is used to join two relations over a condition that includes at least one time attribute of each relation. These attributes usually hold interval information (Soo et al., 1994a) rather than atomic values as in snapshot databases. The variety of temporal joins is therefore given by the possible relationships that can exist between two intervals. Allen identified 7 of these (Allen, 1983). This implies 7 basic variations of the temporal join each determined by the temporal predicate it involves: contain-join, overlap-join, meet-join etc. It can be assumed that the contain-, overlap- and equi-join are the most frequent. The respective predicates, *contain*, *overlap* and *equal*, can be subsumed under the *intersection* relationship. This means that whenever two intervals intersect then either one contains the other or they overlap or they are equal.

Intersection can therefore be used as a global predicate for the three subordinated ones. This is quite useful because partitioning can be done over the intersection condition. For example, the data can be partitioned such that each fragment contains the data intersecting with a certain range[7]. This guarantees that each fragment contains the tuples whose corresponding intervals can possibly intersect and therefore only those than can be possibly joined; tuples that are not present in the same fragment do not satisfy the join condition.

The problem of this method is that an interval can intersect with more than one range and must therefore be present in several of the partition fragments. This means that data has to be replicated which causes an overhead, not only because of the replication process itself but also because of additional processing costs[8]. *This effect does not exist in conventional relational databases that work on atomic attribute domains*[9]. More precisely: for temporal databases the costs of a partitioning algorithm, given by equation (2), might be significantly increased in each component:

- $sc(n)$, the splitting/partitioning costs, by the replication costs (i.e. essentially the additional communication costs),
- $max_{i=1}^{m}(c(n_i))$, the worst case processing costs, by increased values n_i, and

[6] i.e. transaction or valid time

[7] See example of section 2.3.3; range partitioning is probably the most frequently used method. The others are round-robin partitioning (which does not work for the join) and hash-partitioning (see e.g. (4) in section 2.3.2) which is similar to range partitioning.

[8] Apart from the higher number of tuples to be processed the replication can cause duplicates in the result which might have to be eliminated (depending on the operational context)

[9] Almost every relational database management system requires the relations to be normalised to 1NF which means that all attributes have atomic domains.

- $mc(n)$, the merging costs, by the elimination of duplicates in the result.

The only papers – to my knowledge – that discuss data partitioning for temporal joins either assume a partition to be created in a non-specified way and discuss how the problem of replicates can be limited through semantic optimisation(Leung and Muntz, 1992) or can drive around the overhead problem by restricting their technique to sequential machines (Soo et al., 1994b) or have to access partitions various times (Lu et al., 1994). Another paper (Gunadhi and Segev, 1993) on indexing over time attributes meets the interval partitioning problem when balancing the tree structure and discusses only three very simple heuristics for partitioning the interval data. The partitioning problem with respect to index structures is discussed in the following section.

What applies to intervals in transaction or valid time databases certainly applies to rectangles in bitemporal databases. Bitemporal rectangles are subsets of the two-dimensional time space *valid time* $\times$ *transaction time*. Without going into details it is obvious that partitioning of rectangles is even a more complex problem than that for intervals.

3.4 Related Research Areas

There are two research areas that are also confronted with the problem of partitioning intervals and rectangles. It is therefore useful and beneficial to consider and watch proposals made in these areas:

Indexes for Temporal Data: (Kolovson, 1993) and (Gunadhi and Segev, 1993) propose index structures over interval data, called *segment index* and *AT-index* respectively. They use B^+-tree-like techniques for building and maintaining the index. Balancing these trees and achieving minimal redundancy require partitioning the time space such that a minimal number of intervals overlap the partition points.

This problem is simpler than the interval partitioning problem encountered in parallel temporal databases: here only the number of overlapping intervals has to be minimised not the possibly complex cost function describing the overhead caused by a certain number and a certain distribution of overlaps.

Gunadhi and Segev propose very simple heuristics which they consider as a first approach; they finally state: "For future research, we would like to explore further the issue of time partitioning, primarily in the context of multidimensional search structures."

Spatial Databases: Spatial data is comparable to temporal in many respects but in particular in the fact that it is non-scalar: frequent data entities are polygons, rectangles, squares, intervals etc. A lot of problems caused by the non-scalarity of these objects are therefore in common with temporal databases. Amongst these is also the problem of partitioning the geometric space for the purpose of parallel processing or spatial indexing. Despite this similarity, many authors, e.g. in (Leung and Muntz, 1993), (Kolovson, 1993), (Gunadhi and Segev, 1993), point to the different semantics of time with respect to space. Geometric objects have, for example, no natural clustering tendencies such as temporal intervals. Another example is the monotony, in particular of transaction time[10]. These properties allow many interesting optimisations that are not possible in spatial data processing.

[10]This refers to the fact that subsequently inserted tuples hold increased values in their transaction time attribute.

4 Work done so far

Building on my experience with parallel database research through (Zurek and Thanisch, 1993), (Zurek et al., 1995) and (Zurek and Thanisch, 1995) my main efforts during the first year concentrated on getting familiar with the research area of temporal databases and its parallel issues. Naturally these activities were widely spread and not driven by one particular research problem but by several minor aspects. The following gives a list of issues that I have been working on:

Literature Review: As I mentioned in section 3, the group of leading researchers in the temporal database area are currently working on a common framework. This has led to several common publications by this group, in particular (Tansel et al., 1993), papers regarding the TSQL2 standard (Jensen et al., 1994a), (Snodgrass et al., 1994a), (Soo et al., 1994a), (Snodgrass et al., 1994c), (Snodgrass et al., 1994b), (Jensen et al., 1994b) etc. and papers structuring the research area like (Jensen et al., 1992) and (Kline, 1993). These publications seemed a good starting point to me.

After that I concentrated on papers discussing temporal processing and implementation issues like (Soo et al., 1994a), (Tuzhilin and Clifford, 1990), (Sarda, 1993), (Soo et al., 1994b), (Gunadhi and Segev, 1991), (Leung and Muntz, 1990), (Leung and Muntz, 1992), (Leung and Muntz, 1993). The only paper drawing a relation between temporal and parallel relational databases is (Leung and Muntz, 1992) (with parts of it published in (Leung and Muntz, 1993)). I therefore contacted the authors, Leung and Muntz, asking them for further work performed in this area. Both replied that they would not know of any further research done on such issues and that they were concentrating on the standardisation of TSQL2 and not seeking to work on the issues that were mentioned in their paper, at least not in the near future.

I reviewed more specific literature while working on the following issues.

Generating Temporal Interval Data: I implemented some programs to generate random temporal interval data of certain probabilistic characteristics. These programs implemented similar methods to those that are frequently used for modelling effects like data skew in parallel databases. This 'exercise' gave me a good idea of how temporal interval data can be distributed over the time space and that there is good reason to assume effects like temporal data skew.

These programs might serve in the future as part of a testbed for analysing and evaluating new algorithms and partitioning strategies for temporal databases.

Data Partitioning for Relational Database Operations: I studied this issue in some more detail (Zurek, 1995). Results of this study were presented in sections 2.3 and 3.3.

Temporal Join: In order to get to know about the specific characteristics of the temporal join I looked at papers (Soo et al., 1994b), (Gunadhi and Segev, 1991) and (Leung and Muntz, 1992) in some more detail. As each of the authors assumes another join condition I tried to figure out how the full bandwidth of temporal joins can be defined. And in fact, Allen defines all possible relationships between temporal intervals in a paper (Allen, 1983) focusing on temporal knowledge representation issues. Allen's interval relations therefore imply the whole bunch of basic temporal join variations. This allows to classify temporal joins and to separate those that cannot be implemented by using the same data partitioning strategy.

Interval Partitioning Problem: I dedicated a marginal amount of time to determine the complexity of the interval partitioning problem. Actually, I suspected its basic version already to be NP-complete and studied several publications on other partitioning problems like (Garey and Johnson, 1979), (Balcázar et al., 1988), (Hyafil and Rivest, 1973), (Kernighan and Lin, 1970) or (Zhou, 1993).

The basic interval partitioning problem can be mapped to an equivalent problem in graph theory, the sequential graph partitioning problem. The latter was tackled at the beginning of the 1970s in the context of code placement on subsequent pages in memory. Kernighan gave an optimal algorithm for the basic sequential graph partitioning problem in (Kernighan, 1971). However, this dynamic programming algorithm relies on simple cost functions for the partition. A significant open issue is whether more sophisticated cost functions might change the complexity of the problem.

The practical applicability of Kernighan's algorithm is restricted because of its poor running time performance, even when assuming simple cost functions. Nevertheless, it constitutes an important theoretical result and can be used to compute optimal partitions in order to compare them with those derived from fast-running heuristics. Its dynamic programming background might also serve as an initial step towards creating incremental strategies for creating partitions, i.e. whenever a temporal relation is updated its (already existing) partition can be updated too by using an incremental strategy.

Rectangle Partitioning Problem: The bitemporal counterpart of the interval partitioning problem adds a new dimension towards partitioning. In contrast to the interval case, it is not possible to define any suitable order of the nodes in the graph that result from a mapping of the rectangle problem. Such an order is a prerequisite for sequential graph partitioning.

Rectangle graph partitioning is equivalent to partitioning planar graphs of maximum degree 4. Planar graph partitioning is still an open problem in complexity theorie. At the first sight, the restriction to a maximum degree of 4 does not seem to make the problem much easier.

These facts suggest that there is no easy way to determine the complexity of this problem and that it probably goes beyond the goal of my PhD thesis to investigate this question in more detail.

Classification of Partitioning Strategies: I created a rudimentary taxonomy of partitioning strategies depending on properties like dynamic – static partitioning, whether they are based on graphs or statistical, use complete, sample or meta information, are incremental or not, heuristical or optimal.

A Simple Statistical Partitioning Approach: Assuming uniformity of the interval data I was able to derive a formula to calculate the number of fragments of an optimal partition. This formula determines the degree of parallelism that is used in the ideal case. This might be a very useful result as can be used as an approximation for non-uniform cases. This has to be confirmed by experiments.

I considered further degenerated cases of the interval partitioning problem. As in the case of uniformity these cases might help to derive further approximations.

Cost Model Considerations: In order to get an idea in which way the replication of tuples causes an overhead I ran preliminary tests on a Connection Machine CM-200. They show some characteristics of the machine's communication network. Naturally, these are very much hardware-dependent results that do not apply to other parallel machines. Nevertheless, one has to know these facts and to identify certain common properties for creating a proper model for the overhead costs.

Considerations on MPI: PVM and MPI emerge to be future parallel programming standards. Both define virtual parallel machines which could be used as a parallel model to underly my research. I therefore attended a course on programming with MPI at EPCC.

5 The PhD Proposal

In sections 2 and 3, I introduced parallel and temporal databases. In order to successfully combine these two areas one has to find efficient methods to partition temporal data on their temporal attribute(s). For the reasons outlined in sections 2 and 3, I will concentrate on *temporal data partitioning for intra-operator parallelism.*

5.1 Analytical Part

The analytical part of the project comprises a thorough theoretical analysis of the complexity of this problem. This has already been completed. The results show that is not trivial (see section 4). The problem can be solved in polynomial time for temporal intervals and for very simple cost functions by an algorithm suggested in (Kernighan, 1971). In praxis, this algorithm is not feasible for the following reasons:

- The Kernighan algorithm, adapted to the interval partitioning problem, has a time complexity of $O(n^2)$ where n is the number of interval start- and endpoints occurring in the temporal relation. Temporal relations can be expected to hold millions of tuples probably involving several hundreds of thousands of different start- and endpoints. n is therefore far too large in order to get a feasible performance.
- Cost functions modeling real-word computational behaviour are too complex for adjusting the algorithm. This is just not possible.
- If the (probably huge) graph that corresponds to a temporal relation is not kept as meta data, e.g. in the database catalog, the algorithm would need to scan the entire temporal relation in order to create an optimal partition. This would not only annihilate any benefit drawn from parallelism but easily be outperformed by standard sequential processing.

There is a further aspect: as stated in section 2.3.3, there are two basic levels of data partitioning: main memory and secondary storage. In main memory, partitioning requires dynamic, 'on-the-spot' strategies which can react to run-time information, e.g. in the case of a join that requires a unique partition for the relations involved in the join. In secondary storage a more static approach is feasible: an existing partition should provide a good load balance between the disks. New tuples are usually inserted into the temporal relation in an append-manner due to the monotony of time. Instead of recalculating a suitable partition – like the Kernighan-algorithm – it is better to incrementally adjust the existing partition in order to maintain the load balance. This means that – essentially – two groups of partitioning strategies are required: 'on-the-spot' and incremental techniques.

5.2 Synthetical Part

In the synthetical part of my PhD project, I want to investigate heuristics and design such strategies for obtaining good data partitions for temporal data, occurring in temporal and bitemporal database relations.

I suppose that such heuristics and strategies can be found by analysing statistical meta-data or data samples. Meta-data is usually stored in the database catalog and is maintained by the database system. This meta-data can then be used by a (parallel) query optimiser in the optimising process. Drawing data samples of a relation can be an alternative to this approach. In fact several authors, like (Soo et al., 1994b), (DeWitt et al., 1991) or (Soloviev, 1993), use data samples for optimising their join algorithms using the Kolmogorov test statistic (Conover, 1980).

Additionally, the semantics of a temporal query can be used to optimise the strategies, e.g. by reducing the number of tuples that have to be replicated during the partitioning stage. A preliminary study of such a semantically optimising approach is given by (Leung and Muntz, 1992).

As an immediate result of this research I expect to get a variety of strategies for creating good temporal interval partitions based on cost models. This knowledge is the foundation for the efficient application of parallel algorithms for temporal operators and can be used for building a query optimiser module for a parallel temporal database system. As a secondary result the research will improve the knowledge and comprehension of the statistical behaviour of temporal data. I can even expect to get statistical models of temporal data which can be used in future projects.

6 Stages, Methods and Timetable of the Project

This section presents the project's stages (section 6.1), the methods and techniques that I intend to apply (section 6.2) and gives a rudimentary timetable at the end (section refs:timetable).

6.1 Stages

The following flow chart shows the stages to be passed for completing the research project.

1. Definition of the (temporal) interval and rectangle partitioning problems (IP, RP).

$\downarrow$

2. Theoretical analysis of the complexities of IP and RP. Rem.: This stage has already been completed and shown that there is no trivial, straightforward solution which would make further research sense- and purposeless.

$\downarrow$

3. Definition of a cost model on which the algorithm design is based and for which partitions are to be created (see discussion on parallel database architectures in section 2.4).

$\downarrow$

4. Identification of the properties of a partition that determine the costs, like the total number of tuples or the maximum number of tuples to be replicated per processing node or the distribution of such tuples over the partition fragments etc.

$\swarrow$ $\searrow$

5.1 Design of temporal data partitioning strategies possibly building on heuristics or drawing data samples. These strategies should optimise the costs implied by the cost model of stage 3.

$\longleftrightarrow$

6.1 Identify statistical parameters of temporal data

(a) that match the properties found in stage 4;

(b) that indicate a strong possibility or tendency towards these properties.

These parameters might not be the usual parameters like standard variation, medium etc.

$\downarrow$ (5.1 → 5.2)

5.2 Evaluation of the strategies using temporal sample data.

$\downarrow$

5.3 (This is an optional stage) Implementation and evaluation on a parallel platform.

$\downarrow$ (6.1 → 6.2)

6.2 Evaluation of case (b) by experiments using temporal sample data.

$\searrow$ $\swarrow$

7. Writing up of the results; producing the PhD thesis.

6.2 Methods & Techniques

The following gives a list of methods and techniques that are relevant to the respective stage. The numbering of section 6.1 will be used.

1,2. These will require methods and techniques of complexity and algorithm theory. Basic literature herefore is (Garey and Johnson, 1979), (Balcázar et al., 1988), (Aho et al., 1983), (Sedgewick, 1988) and many others. Of particular interest are issues and algorithms of graph theory.

3. Models that are used in papers covering data partitioning related issues may serve as a starting point. Examples can be found in (DeWitt et al., 1991), (Ghandeharizadeh and DeWitt, 1990), (Hua et al., 1993) and (Soloviev, 1993). These and many others are in favour of analytical models based on shared-nothing. The advantage of shared-nothing is that it keeps the models simple[11].

   Commercial parallel database products, however, use shared-disk, shared-memory and hybrid architectures. It might therefore be useful to create a more realistic model based on such approaches. Such a model might be very complex and unsuitable to be used by a query optimiser. It is therefore necessary to find a tradeoff.

   Papers looking at this question from a more practical point of view, like (Valduriez, 1993b) (Hua et al., 1991) or (Baru et al., 1995), should be reviewed for this purpose. Furthermore I expect to get new information from a tutorial presenting the newest parallel database developments at the VLDB 1995 conference in Zürich in September 1995.

4. Stage 3 will result in a cost model suitable for analytical purposes. This will allow to identify the determining cost factors for partitioning algorithms. At least some of the factors will be influenced by the underlying data partition. From these factors one can draw back to the characteristics (of data partitions) that affect the costs and to the way in which they affect them.

5.1 The following gives a list of possible starting points:

   - the Kernighan algorithm which gives an optimal solution for IP (result of stage 2),
   - standard non-temporal partitioning strategies like hash and range partitioning,
   - abstracting non-temporal approaches like (Hua et al., 1993),
   - simple heuristics as in (Gunadhi and Segev, 1993),
   - data sampling approaches as in (Soo et al., 1994b), (DeWitt et al., 1991), (Soloviev, 1993).

   An essential step of the design process will be the consideration of the results of stage 4 with respect to the cost model of stage 3.

   Basic background literature for partitioning algorithms is (Jálá, 1992) and for the Kolmogorov test statistics used in data sampling (Conover, 1980).

5.2 With regard to this stage I can build on previous experience in generating (non-temporal) sample data (Zurek et al., 1995), (Zurek and Thanisch, 1995). (Walton et al., 1991)

[11] Nevertheless, a lot of authors make further simplifying assumptions.

and (Walton et al., 1990) provide information on data skew. Latter is usually simulated through the Zipf distribution (Knuth, 1973).

The semantics of transaction and valid time might be modeled by similar methods as used in traffic theory (Leutzbach, 1988). Temporal behaviour, like the arrival of cars at traffic lights[12], are modeled through the Poisson distribution (Bronstein and Semendjajew, 1987).

Preliminary steps towards the generation of temporal sample data have already been undertaken (see section 4).

5.3 There are several options depending on the parallel platform used in this stage: hardware available through EPCC (Connection Machine CM-200, Cray T3D) or from another HPC site in the UK could be used.

Working with the CM-200 could benefit from my previous experiences (Zurek and Thanisch, 1993) (Zurek et al., 1995). Another advantage is its parallel I/O facility. C* and Prism can be used as a programming platform. Nevertheless, the CM-200 represents an out-of-date technology (SIMD architecture; it does not support general purpose programming models; its communication network; no more machines of this type will be produced). This is a serious disadvantage as performance results are of decreasing interest.

The main advantage of the Cray T3D is the fact that it incorporates up-to-date communication and parallel processing technology. It has a general purpose architecture supporting a variety of parallel programming paradigms (data parallel, message-passing). A disadvantage is the lack of a reasonable parallel I/O facility.

Finally, machines provided by other UK HPC sites might be considered if the EPCC machines work out to be unsuitable.

6.1 This stage is a straightforward continuation of the analysis performed in stage 4. There is a beneficial corelation with stage 5.1: results of 5.1 might contribute and motivate approaches of 6.1 and vice versa.

6.2 See stage 5.2. for methods on creating temporal sample data.

7. Usual tools of text and document processing, in particular LaTeX.

6.3 Timetable

Table 1 gives a rough estimation of the respective duration and dates for completion of the stages. The times given are liable to change with a margin of around one month as it is difficult to estimate the progress of each stage precisely. Some of the stages were combined because technically they should be dealt with together although they are separated from the logical point of view. This is due to their technical or logical relationship (see previous sections).

The timetable offers two reserve margins which might be used in the case that one of the core stages[13] is delayed:

[12] This process seems to be similar to the 'arrival', i.e. insertion, of new data in the database at a certain point in time.

[13] i.e. any stage apart from 5.3.

Stage(s)	**Estimated Duration**	**Anticipated Date of Completion**
1., 2.	4 months	already completed
3., 4.	3 months	end of December 1995
5.1, 6.1	3 (4) months	end of March (April) 1996
5.2, 6.2	6 months	end of October 1996
5.3	3 months	end of December 1996
7.	7 months	end of July 1997

Table 1: A rudimentary timetable for the research project.

- Stage 5.3 is definetely optional as stage 5.2 should already provide the results required. It can be considered as a supplementary stage to underpin stages 5.1 and 5.2. It can therefore be dropped in case of a delay.
- The end of the project is aimed for July 1997. This offers an additional 3 months margin for delays in previous stages and in particular in the writing up. In any case, this period can be used for consolidation purposes.

Shifts of completion dates are also liable in the case that a stage produces an interesting result that is viable for being published and/or submitted to a conference. In this case it will require some time to produce a paper for such a purpose.

References

Agerwala, T., Martin, J., Mirza, J., Sadler, D., Dias, D., and Snir, M. (1995). SP2 system architecture. *IBM Systems Journal*, 34(2):152–184.

Aho, A., Hopcroft, J., and Ullman, J. (1983). *Data Structures and Algorithms*. Addison-Wesley.

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.

Bergsten, B., Couprie, M., and Valduriez, P. (1991). Prototyping DBS3, a Shared-Memory Parallel Database System. In *Proceedings of the 1st International Conference on Parallel Distributed Information Systems, Miami Beach, FL, USA*, pages 226–234.

Bergsten, B., Couprie, M., and Valduriez, P. (1993). Overview of Parallel Architectures for Databases. *The Computer Journal*, 36(8).

Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., and Valduriez, P. (1990). Prototyping Bubba, a Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24.

Bronstein, I. and Semendjajew, K. (1987). *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 23rd edition.

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.

Conover, W. (1980). *Practical Nonparametric Statistics*. John Wiley & Sons, New York, 2nd edition.

Date, C. (1995). *An Introduction to Database Systems*, volume I. Addison-Wesley, 6th edition.

DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H., and Rasmussen, R. (1990). The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62.

DeWitt, D. and Gray, J. (1992). Parallel Database Systems: The Future of High Performance 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.

EDS Database Group (1990). EDS – Collaborating for a High-Performance Parallel Relational Database. ESPRIT Conference, Brussels, Belgium.

Englert, S., Gray, J., Kocher, T., and Shah, P. (1989). A Benchmark of Nonstop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases. Technical Report 89.4, Tandem Computer Inc.

Garey, M. and Johnson, D. (1979). *Computer and Intractability – a Guide to the Theory of NP-Completeness*. Freeman.

Ghandeharizadeh, S. and DeWitt, W. (1990). Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. In *Proc. of the 16th International Conference on Very Large Data Bases (VLDB), Brisbane, Australia*, pages 481–492.

Graefe, G. (1990). Encapsulation of Parallelism in the Volcano Query Processing System. *SIGMOD Record*, 19(2):102–111.

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.

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.

Hua, K., Lee, C., and Young, H. (1993). Data Partitioning for Multicomputer Database Systems: A Cell-Based Approach. *Information Systems*, 18(5):329–342.

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.

ISO92 (1992). International Organization for Standardization / International Electrotechnical Commission – Database Language SQL. ISO/IEC 9075.

Jálá, J. (1992). *An Introduction to Parallel Algorithms*. Addison-Wesley.

Jensen, C., Clifford, J., Gadia, S., Segev, A., and Snodgrass, R. (1992). A Glossary of Temporal Database Concepts. *SIGMOD Record*, 21(3).

Jensen, C., Snodgrass, R., and Soo, M. (1994a). The TSQL2 Data Model. A TSQL2 Commentary. available from `ftp.cs.arizona.edu`.

Jensen, C., Soo, M., and Snodgrass, R. (1994b). Unifying Temporal Data Models via a Conceptual Model. *Information Systems*, 19(7):513–547.

Jones, S., Mason, P., and Stamper, R. (1979). Legol 2.0: A Relational Specification Language for Complex Rules. *Information Systems*, 4(4):293–305.

Kernighan, B. (1971). Optimal Sequential Partitions of Graphs. *Journal of the ACM*, 18(1):34–40.

Kernighan, B. and Lin, S. (1970). An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(2):291–307.

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. (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, A. and *et al.* , editors, *Temporal Databases*, chapter 17, pages 418–432. Benjamin/Cummings.

Korth, H. and Silberschatz, A. (1991). *Database System Concepts*. McGraw-Hill, 2nd edition.

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, A. and *et al.* , editors, *Temporal Databases*, chapter 14, pages 329–355. Benjamin/Cummings.

Leutzbach, W. (1988). *Introduction to the Theory of Traffic Flow*. Springer Verlag.

Lockemann, P., Krüger, G., and Krumm, H. (1993). *Telekommunikation und Datenhaltung*. Studienbücher der Informatik. Hanser Verlag.

Lockemann, P. and Schmidt, J., editors (1987). *Datenbank-Handbuch*. Springer.

Lorie, R., Daudenarde, J.-J., Hallmark, G., Stamos, J., and Young, H. (1989). Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience. *IEEE Data Engineering Bulletin*, 12(1):2–8.

Lu, H., Ooi, B., and Tan, K.-L. (1994). On Spatially Partitioned Temporal Join. Technical Report TRC 1/94, Department of Information Systems and Computer Science, National University of Singapore.

Nievergelt, J. (1974). Binary Search Trees and File Organization. *ACM Computing Surveys*, 6(3):195–207.

Parker, D., Muntz, R., and Chau, H. (1989). The Tangram stream query processing system. In *Proceedings of the International Conference on Data Engineering, Los Angeles, CA, USA*, pages 556–563.

Pissinou, N., Snodgrass, T., 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. available from `ftp.cs.arizona.edu`.

Sarda, N. (1993). HSQL: A Historical Query Language. In Tansel, A. and *et al.* , editors, *Temporal Databases*, chapter 5, pages 110–140. Benjamin/Cummings.

Sedgewick, R. (1988). *Algorithms*. Addison-Wesley, 2nd edition.

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. (1994a). TSQL2 Language Specification. *SIGMOD RECORD*, 23(1):65–86.

Snodgrass, R., Ahn, I., Ariav, G., Bayer, P., Clifford, J., Dyreson, C., Grandi, F., Hermosilla, L., Jensen, C., Käfer, W., Kline, N., Leung, T., Lorentzos, N., Mitsopoulos, Y., Roddick, J., Soo, M., and Sripada, S. (1994b). An Evaluation of TSQL2. A TSQL2 Commentary. available from `ftp.cs.arizona.edu`.

Snodgrass, R., Ahn, I., 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. (1994c). A TSQL2 Tutorial. *SIGMOD RECORD*, 23(3):27–33. also available from `ftp.cs.arizona.edu`.

Soloviev, V. (1993). A Truncating Hash Algorithm for Processing Bamd-Join Queries. In *Proc. of the 9th International Conference on Data Engineering, Vienna, Austria*, pages 419–427.

Soo, M., Jensen, C., and Snodgrass, R. (1994a). An Algebra for TSQL2. A TSQL2 Commentary. available from `ftp.cs.arizona.edu`.

Soo, M., Snodgrass, R., and Jensen, C. (1994b). 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., Katz, R., Patterson, D., and Ousterhout, J. (1988). The Design of XPRS. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB), Los Angeles, CA, USA*, pages 318–330.

Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., and Snodgrass, R. (1993). *Temporal Databases – Theory, Design and Implementation.* Benjamin/Cummings.

Teradata Corporation (1985). *DBC/1012 Database Computer System Manual Release 2.0.* Document No. C10-0001-02.

Tuzhilin, A. and Clifford, J. (1990). A Temporal Relational Algebra as a Basis for Temporal Relational Completeness. In *Proc. of the 16th International Conference on Very Large Data Bases, Brisbane, Australia*, pages 13–23.

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. and Dale, A. (1991). Data Skew and the Scalability of Parallel Joins. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 44–51. IEEE Comput. Soc. Press.

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.

Walton, C., Pinsonneault, M., and Haddix, F. (1990). Measurements of Data Skew in Two Databases. Technical Report TR-90-32, Department of Computer Science, University of Texas at Austin.

Wiederhold, G., Fries, J., and Weyl, S. (1975). Structured Organization of Clinical Data Bases. In *Proceedings of NCC*, AFIPS, pages 479–485.

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.

Zurek, T. (1995). A Study of Data Partitioning for Relational Databases. unpublished.

Zurek, T., Minty, E., and Thanisch, P. (1995). Recursive Query Processing on a Connection Machine. In Keane, J., editor, *Parallel Information Processing.* UNICOM Applied Information Technology Series, Alfred Waller Publishers.

Zurek, T. and Thanisch, P. (1993). Processing Linear Recursive Database Queries on the Connection Machine. Technical Report EPCC-TR93-05, Edinburgh Parallel Computing Centre (EPCC).

Zurek, T. and Thanisch, P. (1995). Optimization Strategies for Parallel Linear Recursive Query Processing. Technical Report ECS-CSG-16-95, Dept. of Computer Science, Edinburgh University.