Title Slide

Parallel Temporal Joins

Thomas Zurek

University of Edinburgh

Slide 1: Overview

The talk is concerned with the **parallel processing of temporal joins**. It introduces the **problem rather than the solution** that is discussed in my conference paper. The talk comprises 3 parts:

1. introduction to
   - what are **temporal relations**?
   - what are **temporal joins**?
   - what's the **difference to equi-joins**?
2. focus on
   - **parallel** join processing
   - and its **implications for temporal joins**
3. **basic idea** behind the two optimisations that we propose.

Overview

1. What are **temporal joins**?
2. What are **parallel** temporal joins?
3. Optimisations

Slide 2: Temporal Joins (1)

This slide introduces some basic technical terms and ideas in the context of temporal databases:

- **temporal relations**: Hamlet, Faust (of a performances in various cities)
- each tuple has a **timestamp** telling when the play is performed
- timestamps are represented as **intervals**

An Example

Relation **Hamlet**

City	Timestamp
Paris	[3,4]
Berlin	[3,5]
Rom	[4,6]
Vienna	[8,8]
Oslo	[8,9]
Berlin	[8,9]
...	

Relation **Faust**

City	Timestamp
Berlin	[1,3]
Oslo	[1,3]
Paris	[3,3]
Vienna	[6,7]
Miami	[6,9]
Paris	[7,9]
...	

Slide 3: Temporal Joins (2)

This slide shows two queries involving the two temporal relations of the previous slide:

- the first one implies an **equi-join**,
- the second one implies a **temporal join**.

Equi-joins have a join condition based on an equality predicate. According to Valduriez, over 90% of the join queries (in conventional relational databases) are equi-join queries, e.g. all '*key* = *foreign key*' queries.

Temporal joins have a join condition based on a temporal predicate over the timestamps. In this case, the predicate is temporal intersection of two intervals, i.e. the procedural expression for simultaneity.

Now, we want to focus on the **problem of join processing**.

Hamlet $\bowtie_E$ **Faust** with $E \equiv$ Hamlet.City = Faust.City

Hamlet $\bowtie_E$ **Faust**			
Berlin	[3,5]	Berlin	[1,3]
Vienna	[8,8]	Vienna	[6,7]
Oslo	[8,9]	Oslo	[1,3]
...			

Hamlet $\bowtie_T$ **Faust** with $T \equiv$ Hamlet.Timestamp *intersects* Faust.Timestamp

Hamlet $\bowtie_T$ **Faust**			
Paris	[3,4]	Berlin	[1,3]
Paris	[3,4]	Oslo	[1,3]
Paris	[3,4]	Paris	[3,3]
...			

Slide 4: Temporal Joins (3)

This figure illustrates **an example for processing an equi-join with a nested-loops join algorithm**. I will use this kind of illustration in the remainder of the talk.

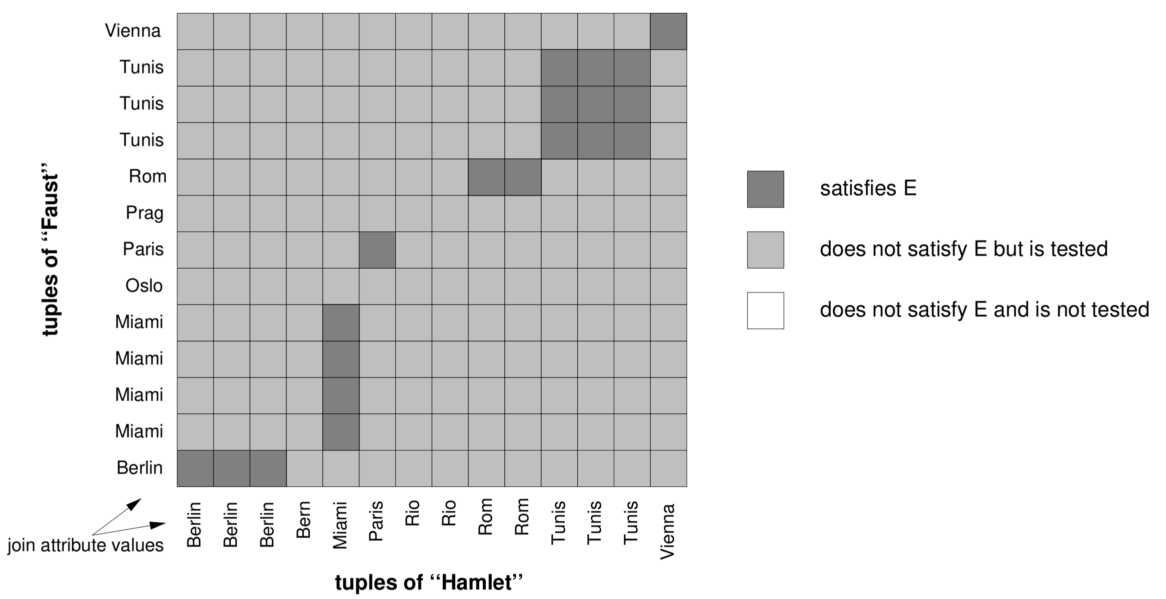
- tuples of HAMLET are represented by **columns**
- tuples of FAUST are represented by **rows**
- columns and rows are **labeled** with the join attribute value of the respective tuple
- each **square** represents a possible tuple combination for the join result.

The **colouring of the squares** shows the way in which the join is processed:

- A **light grey square** shows that the respective tuple combination is tested but does not satisfy the join condition E.
- A **dark grey square** shows that the respective tuple combination is tested and does satisfy the join condition E.
- An **empty / white square** will show that the respective tuple combination is not tested.

A naive nested-loops approach tests all possible combinations ($\rightarrow$ no white squares).

Nested-Loops Equi-Join **Hamlet** $\bowtie_E$ **Faust**



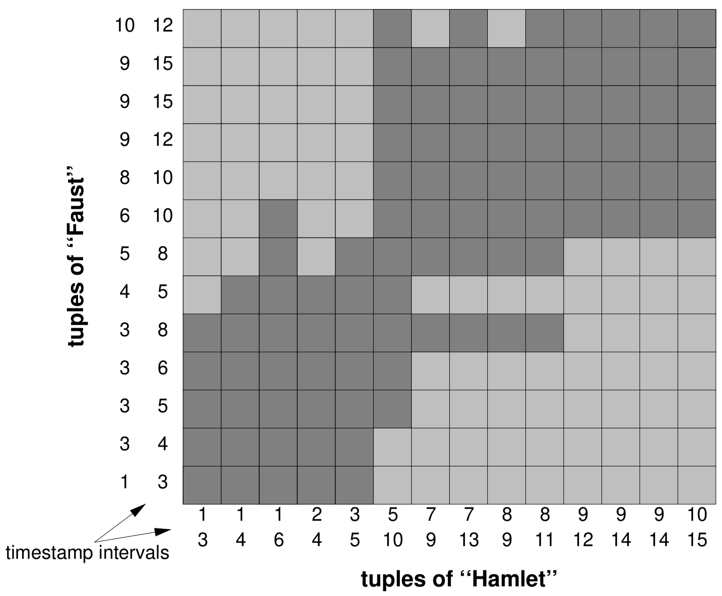
Slide 5: Temporal Joins (4)

This is a similar illustration as on the previous slide but for **a temporal join example** involving similarly sized relations.

Although this is only an example one can observe the frequently arising fact that temporal joins have **a higher selectivity factor** than equi-joins. In other words: temporal join results are usually bigger than results of equi-joins that involve similarly sized relations.

Now we want to turn to **parallel join processing techniques**.

Nested-Loops Temporal Join **Hamlet** $\bowtie_T$ **Faust**



Slide 6: Parallel Temporal Joins (1)

This slide illustrates the way in which **an equi-join can be processed in parallel**.

- **symmetric partitioning** : parallel join processing is usually based on i.e. all relations are partitioned according to the same policy

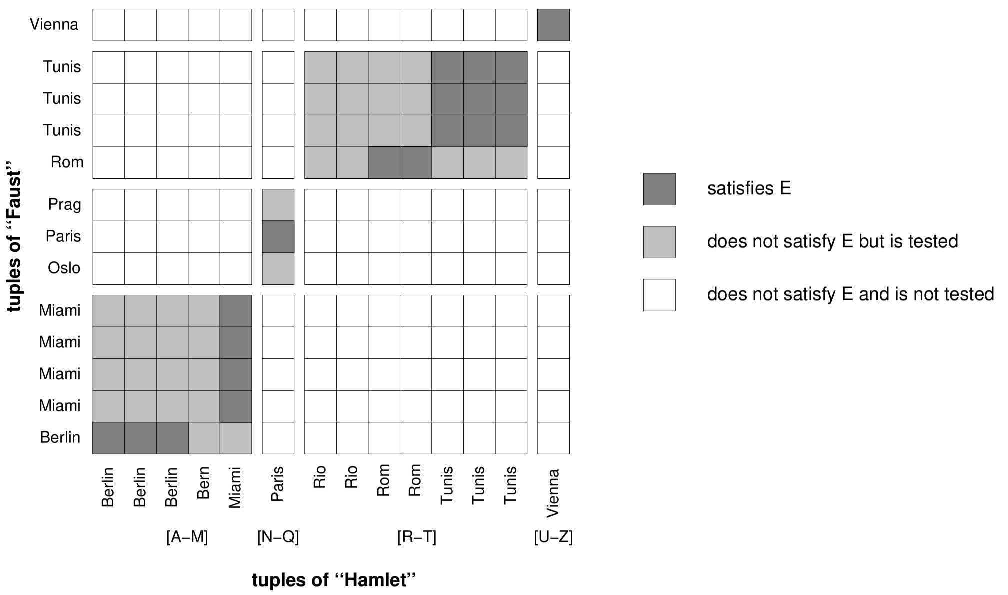
- **one big join** is partitioned into **several small joins**

- **partitioning is based on the join attribute**: tuples that can possibly join are gathered in corresponding relation fragments

In this example, the **partial joins are processed by nested-loops** which is not so naive anymore as the 'success rate' (selectivity factor) of a partial join is much higher than that of the big join.

Let's see what happens when this technique is applied to temporal joins

Parallel Nested-Loops Equi-Join **Hamlet** $\bowtie_E$ **Faust**



Slide 7: Parallel Temporal Joins (2)

This slide shows how the **temporal join example is processed in parallel**. The interesting part lies within **the partitioning process**:

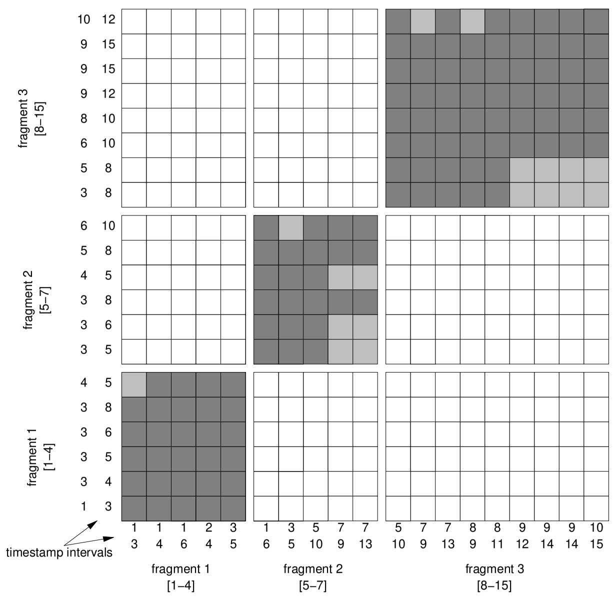
- tuples are partitioned according to their interval timestamp values
- some tuples have to be **replicated**
- as a consequence, the **sizes of the partial joins grow**

We observe the following problem:

tuple replication → replication of tuple comparisons

The **choice of the partitioning ranges** is crucial as it determines **which and how many tuples are replicated**. This is an issue that is not discussed in the paper but we are currently working on that problem.

Parallel Temporal Join **Hamlet** $\bowtie_T$ **Faust**



Slide 8: Parallel Temporal Joins (3)

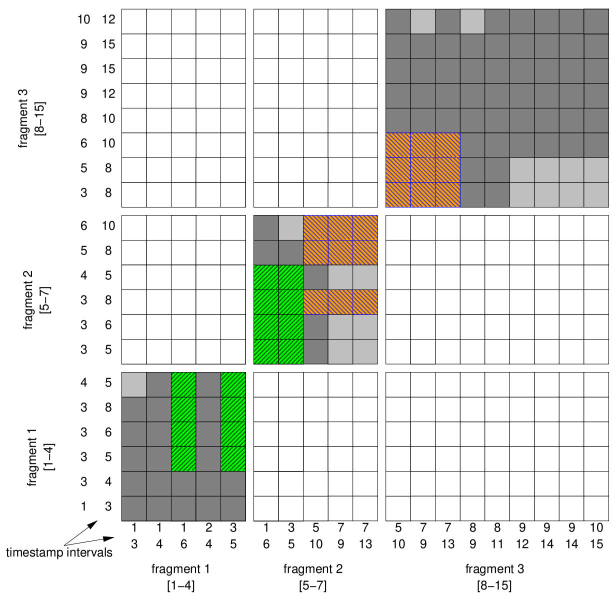
This slide shows the same parallel temporal join example as the previous one but **replicated comparisons are highlighted**. They cause an overhead because

- they lead to **duplicates** in the result (which might have to be removed by expensive process),
- they lead to a **processing overhead** because they are done more than once.

My paper shows how to avoid these replicated comparisons

The basic idea is shown on the next slide which concentrates on the partial join **Hamlet**$_3 \bowtie_T$ **Faust**$_3$ (upper right corner).

Parallel Temporal Join **Hamlet** $\bowtie_T$ **Faust**



Slide 9: Optimisations (1)

The basic idea is to **divide a fragment into two parts**:

- Tuples whose timestamp startpoint falls into the corresponding partitioning range; these tuples do not appear in preceding fragments. $\rightarrow H_3', F_3'$

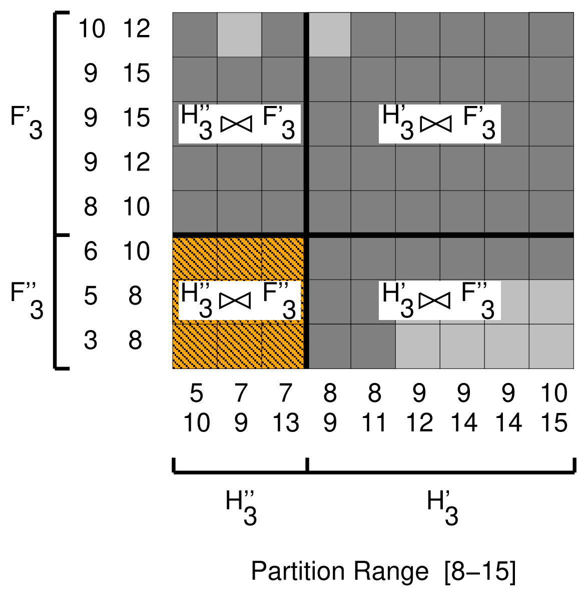
- Tuples whose timestamp startpoint does not fall into the corresponding partitioning range; these tuples do appear in preceding fragments. $\rightarrow H_3'', F_3''$

This allows **two optimisations**:

1. The subjoin $H_3'' \bowtie_T F_3''$ does not need to be computed.
   $\rightarrow$ no duplicates, no processing overhead

2. The sizes of the subfragments H_3', F_3' are easier to control during partitioning: **each tuple appears only in one 'primed fragment'**. For example, one can easily guarantee that H_3' fits into main memory. This reduces the number of disk accesses.

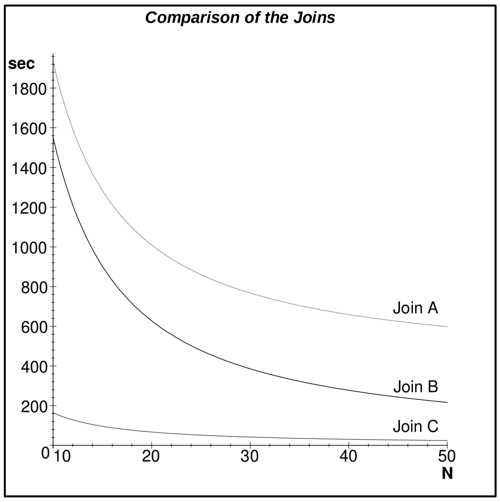
The Partial Temporal Join **Hamlet**$_3$ $\bowtie_T$ **Faust**$_3$



Slide 10: Optimisations (2)

This slide shows the quantitative impact of the optimisations: the response time for three different join computations are shown running on machines with N processing nodes.

Quantitative Impact



- join A: **none** of the optimisations
- join B: **first** optimisation only
- join C: **both** optimisations