Optimisation of Partitioned Temporal Joins

Thomas Zurek
Department of Computer Science
University of Edinburgh
Email: tz@dcs.ed.ac.uk

Abstract. Partitioning data for temporal join processing is not trivial because tuples have to be replicated between data fragments. This causes three types of overheads: (a) an overhead caused by the replication process itself, (b) a processing overhead caused by the additional joining that has to be done and (c) an overhead for removing duplicates in the result. Previous work has mainly concentrated on avoiding (a) but still suffers from the consequences of (b) and (c).

In this paper, we show how partitioned temporal joins can be optimised for sequential and parallel processing by reducing tuple replication, thereby reducing the total overhead. For that purpose, a new data structure, namely the IP-table, is introduced. The idea is to have IP-tables of individual temporal relations stored in the database system’s catalog from which they can be retrieved for the optimisation process. IP-tables of two or more temporal relations might be required for optimisations, too. These can be created by merging IP-tables of individual relations at optimisation time – a fast and straightforward process.

IP-tables can be used for creating and analysing partitions over interval timestamps. Three strategies for partitioning interval data are presented, each of which can be easily and efficiently implemented using IP-tables. The performance determining parameters of a partition can also be derived from IP-tables.

Keywords: temporal join, parallel join, optimisation, interval partitioning.

1 Introduction

Recent years have seen a significantly increasing interest by the industrial and academic community in modeling and storing data that changes over time. One important example is trend analysis, e.g. in the context of data warehouses [7]. Temporal databases have also been the focus of a large number of research papers [9]. With this development comes a whole bunch of new problems: temporal data models, temporal query languages, temporal operators etc. are required. In this paper, we look at one particular performance-critical temporal operator, namely the temporal join, and ways in which temporal join processing can be optimised. We thereby concentrate purely on temporal-specific optimisation and regard our proposal as an enhancement of existing relational optimisation techniques.
In the past, very efficient algorithms have been developed for equi-joins, i.e., joins with an equality join condition. These are based on several forms of **partitioning** [5]. Explicit partitioning – as one form that is, for example, used in hash or parallel joins – breaks up the one ‘big’ join operation into several smaller and independent joins. This can be summarised in the following equation:

\[ R \bowtie C Q = R_1 \bowtie C Q_1 \cup \cdots \cup R_m \bowtie C Q_m \]  

where the \( R_k \) and \( Q_k \) are referred to as fragments of the relations \( R \) and \( Q \), respectively. \( C \) is a boolean expression and said to be the join condition.

(1) 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). In practice, most join conditions in real queries are built on an equality predicate such as “key = foreign key” conditions, as in the case of a customers table being joined with an address table via equal name attribute values.

In the case of **temporal joins**, however, the join conditions are usually based on a non-equality predicate. We adopt the most frequently used temporal data model in which tuples of a temporal relation have an interval timestamp. 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 [1]. **Intersection** of two intervals is the most prominent and the most general join predicate in this case. It is a supertype of most other possible relationships between two intervals. Furthermore, it plays a similarly important role for interval data as equality for atomic data: it establishes the notion of simultaneity of two temporal objects.

A join condition based on intersection of two timestamp intervals, however, imposes a problem on partitioning temporal joins as in (1): it is usually impossible to create disjoint fragments \( R_1, \ldots, R_m \) \( (Q_1, \ldots, Q_m \) respectively) that are disjoint. Consider the example of figure 1. It visually describes the computation of a temporal join \( R \bowtie Q \) that is partitioned into three partial joins. Tuples of \( R \) are put along a horizontal axis, tuples of \( Q \) along a vertical axis. \( R \) and \( Q \) are partitioned into three fragments respectively. Squares symbolise possible combinations of tuples of \( R \) and \( Q \). The figure shows nested-loop computations for the partial joins \( R_k \bowtie Q_k \), i.e., every tuple in \( R_k \) is compared with every tuple in \( Q_k \). Tuple comparisons that are successful are shown in dark grey, comparisons that are performed in a preceding partial join are in black, unsuccessful comparisons in light grey. Figure 1 makes the following problems evident:

(a) Tuples have to be replicated between fragments. This causes an overhead (**replication overhead**). Figure 1 has a total of 13 tuple replications (5 for \( R \), 7 for \( Q \)).

(b) As a consequence, the fragments grow. This implies more tuple comparisons (**processing overhead**). Figure 1 has a total of 132 comparisons. Using different breakpoints of the time domain, e.g., \( \{5, 9\} \) instead of \( \{4, 7\} \) as in figure 1, would lead to 156 comparisons, i.e., 18% more.

(c) Finally, the tuple replications cause duplicates in the result (see black squares). This contributes to the processing overhead but also causes an additional