# An Optimisation Scheme for Coalesce/Valid Time Selection Operator Sequences

Costas Vassilakis
University of Athens, Department of Informatics
Panepistimiopolis, TYPA Buildings
Athens, 15771, Greece
C.Vassilakis@di.uoa.gr

**Abstract:** Queries in temporal databases often employ the *coalesce* operator, either to coalesce results of projections, or data which are not coalesced upon storage. Therefore, the performance and the optimisation schemes utilised for this operator is of major importance for the performance of temporal DBMSs. Insofar, performance studies for various algorithms that implement this operator have been conducted, however, the joint optimisation of the coalesce operator with other algebraic operators that appear in the query execution plan has only received minimal attention. In this paper, we propose a scheme for combining the coalesce operator with selection operators which are applied to the valid time of the tuples produced from a coalescing operation. The proposed scheme aims at reducing the number of tuples that a coalescing operator must process, while at the same time allows the optimiser to exploit temporal indices on the valid time of the data.

**Keywords:** *Temporal databases, Coalescing, Valid time selection, Optimisation, Query Processing*

## 1 Introduction

Queries in temporal databases [24] frequently require the usage of the *coalesce* operator [13] in order to guarantee that resulting information is represented using a minimal number of value-equivalent [13] tuples that have maximal timestamps. As pointed out in [4], uncoalesced relations may arise in many ways, most commonly by applying a non-coalesce preserving operator, such as the *project* or *union* operator, or by not enforcing coalescing upon insertion and update (although most temporal data models, such as [2], [7], [10], [23] enforce such coalescing). Thus, it is crucial for a temporal DBMS to employ efficient algorithms for the implementation of the *coalesce* operator and apply effective optimisation schemes, so as to increase the DBMS throughput and reduce the query response time.

Insofar, the semantics of the coalesce operator (called *fold* in [16] and *compress* in [19]) have been studied ([25], [17]) and various algorithms for its implementation have been proposed and assessed ([16], [19], [4], [5]). In particular [4] and [5] identify 17 algorithms for the implementation of the coalesce operator and evaluate the performance of 10 of them with respect to various parameters that may affect the execution time, such as the reduction factor, the explicit attribute skew and the timestamp skew. The results of these studies may be exploited by the optimiser of a temporal DBMS, in the process of choosing the most appropriate algorithm for each case. Furthermore, [4] and [5] identify query tree reorganisation rules that the optimiser may use to "push" selection predicates below coalescing operators, so as to reduce the number of tuples that the coalescing operators must process. These rules may only be applied in cases that the selection predicate *does not reference* the tuple valid time timestamp, since the tuple valid time timestamp is not actually known until the coalescing operation is performed. For example, in the evaluation of the TSQL2 query

```
select *
from Employee(name, salary)(period) as e
where e.name = 'John'
```

against the table

| Name | Salary | Rank | Valid Time |
|---|---|---|---|
| John | 20000 | Clerk | [1/86-12/87] |
| John | 20000 | Manager | [1/88-12/90] |
| Mary | 21000 | Manager | [1/87-12/89] |

it is possible to apply the restriction *where e.name = 'John'* before the coalesce operator, whereas in the query

```
select *
from Employee(name, salary)(period) as e
where valid(e) contains period '[1/1987–1/1989]'
```

such a rearrangement is not possible, because the first two tuples that do not qualify with respect to the selection predicate will be merged by the coalesce operator into a single tuple that *does* qualify. Hence, if the query designates restrictions only on the valid time resulting from the coalesce operator, the *whole bulk* of the input relation data must be processed by the *coalesce* operator.

In this paper we present a scheme for optimising the execution of *coalesce/valid time selection* operator sequences, where the valid time selection operator references the timestamp resulting from the *coalesce* operator. The goal of this optimisation scheme is to reduce the number of tuples that must be actually processed by the *coalesce* operator by *pre-filtering* tuples that cannot contribute to the final outcome of the *coalesce/valid time selection* operator sequence. The proposed approach may be combined with any algorithm for the implementation of the *coalesce* operator and allows the exploitation of index structures (i.e. queries retrieving tuples having valid times included in a given range–amongst others [1], [3], [8], [12], [15], [18], [20], [21], [22]).

The remainder of this paper is organised as follows: in section 2 the optimisation scheme is described in detail and it is proved that the optimised execution scheme yields the same result with the *coalesce/valid time selection* operator sequence. In section 3 the performance of the optimisation scheme is assessed and compared to the execution time of the non-optimised operator sequence. Experimental results are also presented so as to substantiate the benefits resulting from the proposed approach. Finally, section 4 concludes and outlines future work.

## 2 Optimisation Scheme

Consider the valid table depicted in figure 1, which stores the rank and salary evolution of university staff, and the query "find the names of employees whose salaries changed after the 31st of December 1997". This query may be expressed in TSQL2 [25] as follows:

```
select distinct F.Name
from Faculty(Name, Salary)(PERIOD) as F
where TIMESTAMP '31 DECEMBER 1997' PRECEDES VALID(F)
```

(a timestamp *precedes* a period if all the time points in the period occur *after* the timestamp on the time axis.) In the instance of table *Faculty* illustrated in figure 1, we can intuitively identify three categories of tuples with respect to the select query:

| Name | Rank | Salary | Valid Time | |
|---|---|---|---|---|
| John | Lecturer | 50000 | [01/1997, 08/1997] | (1) |
| John | Lecturer | 60000 | [09/1997, 03/1998] | (2) |
| John | Associate | 60000 | [04/1998, Now] | (3) |
| Jenny | Lecturer | 65000 | [06/1997, 12/1997] | (4) |
| Jenny | Associate | 65000 | [01/1998, Now] | (5) |
| Jack | Associate | 75000 | [08/1997, 11/1997] | (6) |
| Jack | Professor | 80000 | [03/1998, Now] | (7) |

**Figure 1 – The valid time table Faculty**

1. tuples whose timestamp satisfies the given predicate and thus will appear in the final result, *unless* they are coalesced with some other tuple *not satisfying* the predicate (tuples 3, 5 and 7).
2. tuples $t_i$ whose timestamp *does not satisfy* the given predicate, but *may be coalesced* with a tuple *t* from category (1) (tuples 2 and 4). If such a coalescing actually occurs, tuple *t* will not appear in the final result, because it will have been replaced by a value-equivalent tuple whose valid time timestamp does not satisfy the selection predicate.
3. tuples that *neither satisfy* the selection predicate *nor* are bound to be coalesced with any tuple from category (1), because their valid time timestamp is neither overlapping nor adjacent with the time window in which all timestamps of tuples in category (1) fall (tuples 1 and 6).

Generally, a fourth tuple category may be identified, which includes tuples that do not satisfy the selection predicate alone, but *may do so* after being coalesced with other tuples, resulting in predicate satisfaction. This may occur, for example, when using the predicate *contains*.

The optimisation scheme proposed in this paper consists of *filtering out* the tuples in category (3) *before* the coalesce operator is evaluated. The coalesce operator will process the remaining tuples and, finally, *post-filtering* will be applied on the result of the coalesce operator to eliminate tuples from category (2) and tuples from category (1) that do not satisfy the selection predicate after coalescing. The predicate used for post-filtering is identical to the original selection predicate.

More formally, the proposed optimisation scheme transforms each

$$s_{cond(vt)}(\text{coalesce}(R))$$

operation sequence, where *cond(vt)* is a condition involving only the tuple valid time and constants to an operation sequence

$$s_{cond(vt)}(\text{coalesce}(s_{pre\text{-}cond(vt)}(R)))$$

where *pre-cond(vt)* is again a simple condition involving only the tuple valid time and constants. The reason for introducing the extra selection operator is twofold:

1. the number of tuples that must be processed by the *coalesce* operator is reduced. The reduction factor is determined by the selectivity factor of the pre-filtering predicate *pre-cond(vt)*. However, since the complexity of the selection operator is (sub)linear, while the complexity of the coalesce operator is super-linear [4], even low reduction factors result in considerable performance gains. It is also possible to build compound execution algorithms which implement together the pre-filtering and the coalesce operators (or even *both* filtering operators *and* the coalesce operator) as suggested in [9], eliminating any I/O overheads that may be induced from the introduction of an additional operator.
2. the pre-filtering condition *pre-cond(vt)* may be used to retrieve only the relevant tuples through a *temporal index*, in the case that such a structure has been defined on the base relation. This will contribute in reducing the overall cost even more, since the non-relevant portions of the base relation will not be accessed.

| Selection predicate (*cond*(*vt*)) | Pre-filtering predicate (*pre-cond*(*vt*)) |
|---|---|
| instant precedes *vt* | instant <= end(*vt*) |
| instant overlaps *vt* | true |
| *vt* precedes instant | begin(*vt*) <= instant |
| period precedes *vt* | end(*period*) <= end(vt) |
| period = *vt*<br>*vt* = period | end(*vt*) >= begin(period) – 1 granule *and* begin(*vt*) <= end(period) + 1 granule |
| period meets *vt* | end(*vt*) >= end(period) - 1 granule |
| period overlaps *vt*<br>*vt* overlaps period | true |
| period contains *vt* | end(*vt*) >= begin(period) – 1 granule *and* begin(*vt*) <= end(period) + 1 granule |
| *vt* precedes period | begin(*vt*) <= begin(period) |
| *vt* contains period | true |
| begin(*vt*) < instant | true |
| begin(*vt*) = instant | end(*vt*) >= instant – 1 granule |
| begin(*vt*) > instant | end(*vt*) >= instant |
| end(*vt*) < instant | begin(*vt*) <= instant |
| end(*vt*) = instant | begin(*vt*) <= instant + 1 granule |
| end(*vt*) > instant | true |

**Figure 2 – Mapping the selection predicates to *pre-filtering* predicates**

Figure 2 illustrates the mapping between the condition of the selection operator *cond(vt)* and the repsective pre-filtering condition. The operators defined by TSQL2 [25] for comparisons of periods (valid time timestamps) with instants and periods (condition constants) are considered here, and the application of the functions *begin* and *end* [25] to the tuples' valid time is also taken into account. Note that for five conditions this optimisation scheme is not applicable; this is due to the fact that no tuples can be classified in category (3) described above, i.e. in the category of tuples that neither satisfy the selection predicate nor are bound to be coalesced with any tuple which will potentially appear in the final result. In these cases, the pre-filtering predicate is set to *true*. Throughout figure 2, the tuple valid time timestamp is designated as *vt*.

We will now prove that the proposed transformation, i.e. the introduction of the pre-filtering predicate, preserves the result equivalence. For brevity reasons we will contain ourselves to proving the equivalence only for the first predicate mapping (instant precedes *vt*); the result equivalence for the remaining predicate mappings may be proved similarly. Throughout the proof we will use the following notations:

- *t.expl* will denote the explicit attributes of tuple *t*.
- *t.vt* will denote the valid time of tuple *t*.
- *I(p)* will denote the set of instants included in period *p*. This set may not be finite if a continuous model of time is employed.

**Theorem:** For any valid time relation R and a given instant *inst*, it holds that

$$s_{\text{inst precedes } vt}(\text{coalesce}(R)) = s_{\text{inst precedes } vt}(\text{coalesce}(s_{\text{inst} <= \text{end}(vt)}(R)))$$

**Proof:** Let $R_1$ and $R_2$ be the results of the operations $s_{\text{inst precedes } vt}(\text{coalesce}(R))$ and $s_{\text{inst precedes } vt}(\text{coalesce}(s_{\text{inst} <= \text{end}(vt)}(R)))$. We will prove that some tuple $t \in R_1$ if and only if $t \in R_2$. We will prove first that if a tuple *t* appears in $R_1$ then the same tuple *t* appears in $R_2$.

Let $C = \{t_1, t_2, ..., t_n\}$ be the set of tuples of R that are coalesced to produce tuple *t*. By virtue of the definition of the coalesce operator we have that

1. $\forall i \in \{1, 2, \ldots, n\}$ $t.\text{expl} = t_i.\text{expl}$
2. $I(t.vt) = \bigcup_{i=1}^{n} I(t_i.vt)$

Furthermore, since *t* appears in $R_1$, *inst precedes t.vt Þ inst < begin(t.vt)* (due to the definition of the *precedes* predicate [25]). Additionally,

$$\forall i \in \{1, 2, \ldots, n\}\ \text{begin}(t.vt) = \min(I(t.vt)) \leq \text{end}(t_i)$$

thus the pre-filtering predicate is satisfied for every $t_i$. Consequently, all members of the tuple set *C* will participate in the coalesce operation of the transformed operation sequence; additionally, $s_{\text{inst} <= \text{end}(vt)}(R) \subseteq R$, thus $s_{\text{inst} <= \text{end}(vt)}(R)$ may not contain any other tuple that may be coalesced with any $t_i$, hence $t \in$ coalesce($s_{\text{inst} <= \text{end}(vt)}(R)$). Finally, tuple t satisfies the post-filtering predicate (since the post-filtering predicate is identical to the original selection predicate and *t* is known to satisfy the latter), thus *t* will appear in $R_2$.

Now we will prove that if a tuple *t* appears in $R_2$ then the same tuple appears in $R_1$. Let *t* be a tuple appearing in $R_2$. We define the sets *C* and $C_{rejected}$ as follows:

$$C = \{t_i \in R: t.\text{expl} = t_i.\text{expl} \wedge \text{inst} <= \text{end}(t_i.vt)\}$$

$$C_{\text{rejected}} = \{t_i \in R: t.\text{expl} = t_i.\text{expl} \wedge \text{inst} > \text{end}(t_i.vt)\}$$

i.e. C is the set of tuples in R that are value-equivalent to *t* and satisfy the pre-filtering predicate, whereas $C_{\text{rejected}}$ is the set of tuples in R that are value-equivalent to *t* but do not satisfy the pre-filtering predicate. Since $t \in R_2$, *t Î coalesce(C)* (*C* contains precisely the tuples in $s_{\text{inst} <= \text{end}(vt)}(R)$ that may produce *t* via coalescing), and additionally satisfies the post-filtering predicate *inst precedes t.vt Þ inst < begin(t.vt)* (due to the definition of the *precedes* predicate). Now let $t_x$ be a tuple in $C_{rejected}$. By virtue of the definition of $C_{rejected}$, *inst > end($t_x$.vt)*, or, equivalently, *end($t_x$.vt) < inst*. Since

$$\forall i \in \{1, 2, \ldots, n\}\ \text{end}(t_x.vt) < \text{inst} < \text{begin}(t.vt) \leq \text{begin}(t_i.vt)$$

tuple $t_x$ cannot be coalesced with any $t_i$; given that this is true for any $t_x$ *Î* $C_{rejected}$, *t Î coalesce(C È* $C_{rejected}$*)*. We can easily conclude that *t Î coalesce(R)*, since *R - {C È* $C_{rejected}$*}* does not contain any tuple that is value-equivalent to *t*. Finally, given that *t* satisfies the post-filtering predicate *inst precedes vt* (because it appears in $R_2$), *t Î* $s_{\text{inst precedes } vt}$*(coalesce(R))*, which is $R_1$.

We have proved that a tuple *t* belongs to $R_1$ if and only if tuple *t* belongs to $R_2$. Thus, the introduction of the pre-filtering operator does not alter the result of the operation sequence $s_{\text{inst precedes } vt}$*(coalesce(R))*. Similar proofs may be given for the remaining cases illustrated in figure 2.

## 3 Performance Analysis

The optimisation scheme described in the previous section introduces an extra node in the query execution plan, i.e., the selection operator implementing pre-filtering, aiming at decreasing the number of tuples that need to be processed by the coalesce operator. The benefits resulting from this optimisation scheme depend on how the optimisation scheme is implemented and the selectivity factor of the pre-filtering predicate as follows:

1. An extra cost of *N * ts_comp* is always incurred, which corresponds to the cost of determining whether each tuple satisfies the pre-filtering predicate (*N* is the number of tuples in the relation, whereas *ts_comp* is the cost of timestamp comparison). In two cases, this cost may increase to *2 * N * ts_comp*, since the pre-filtering predicate is a conjunction of timestamp comparisons. A portion of this cost may be compensated for, since pre-filtering may result to the production of fewer output tuples by the coalescing

operator, thus post-filtering will be applied to fewer tuples.

2. If pre-filtering selection and coalescing are not merged into a single execution node *and* the execution policy writes the result of each operation to a temporary relation on the disk where it is accessed by the next operator, an extra cost of

$$2 * \left\lceil \frac{N * SF}{\left\lfloor \frac{PageSize}{tupleSize} \right\rfloor} \right\rceil * PageIOCost$$

is added, accounting for the cost of writing the output of the pre-filtering selection to the disk and reading it again, in order to be processed by the coalesce operator. ($0 \leq SF \leq 1$ is the selectivity factor of the pre-filtering predicate, *PageSize* is the size of a disk page, *tupleSize* is the number of bytes within a relation tuple and *PageIOcost* is the cost of reading or writing a disk page). However, if pre-filtering is integrated with the coalescing operation into a single execution node *or* the pre-filtering operator forwards its results to the coalescing operator on a page-by-page (or tuple-by-tuple) basis, this cost will not be incurred.

3. Since, as identified in [4] and [5], the coalescing algorithms are essentially variations of the sort-merge and partitioning algorithms used for duplicate elimination, the complexity of these algorithms in the general case will be O(N* log(N)), [11]. Pre-filtering will reduce the amount of tuples processed by the coalescing operator to $N * SF$, thus the asymptotic complexity of the operator will not be affected; however the overall operation cost will be decreased by a factor which is greater than *(1/SF)*. The experimental results that follow show that this performance gain is considerable, even for selectivity factors that are close to 0.95.

4. Identifying *coalesce/valid time selection* operator sequences in the query execution plan is an extra task for the optimiser, thus an *optimisation cost* should also be accounted for. However, this is expected to be very small, since the optimiser will generally try to "push" selection predicates as low as possible within the execution plan, thus the target temporal selection predicates will always be adjacent to the respective coalescing operators. Hence, a simple traversal of the query execution plan tree will suffice to identify and transform the target operator sequences; moreover, the identification of these sequences may be integrated into the selection "push" procedure.

The optimisation scheme described in the previous section has been evaluated using the *Time-It* testbed [14]. The diagrams that follow illustrate the measurements obtained from comparing the algorithms including pre-filtering to the algorithms not including it. Out of the 10 algorithms included in the performance study of [4] and [5] we present results for only four; the results obtained for the remaining six are similar. The diagrams illustrate the results for the following algorithms:

- partitioning on explicit attributes with hybrid buffer allocation strategy (EP-H).
- sorting on explicit attributes with grace buffer allocation strategy (ES).
- partitioning on the timestamp with grace buffer allocation strategy (TP).
- sorting on the timestamp with hybrid buffer allocation strategy (TS-H).

This subset was chosen so as to include various combinations of basic methods (sorting, partitioning), primary comparison targets (explicit attributes, timestamp) and buffer allocation strategies (grace, hybrid).

Each diagram presents the speedup percentage obtained from building pre-filtering into a specific coalescing algorithm, under various amounts of available main memory and pre-filtering predicate selectivity factors. If *A* is an algorithm for coalescing, $A_{pre\text{-}filter}$ is the same algorithm A extended to include pre-filtering, and *T(X)* is the time needed to execute algorithm *X*, the speedup percentage of algorithm $A_{pre\text{-}filter}$ against algorithm *A* is defined as $\frac{T(A) - T(A_{pre-filter})}{T(A)}$. In all cases, a 16MB relation of 16-byte tuples was used; the tuple's lifespan was set to 10 chronons, while 1000 distinct values were used for the explicit attributes producing approximately 1000 value-equivalent tuples within the relation. We chose a small tuple lifespan so as to use a common test case for all algorithms, since the algorithms based on timestamp sorting do not perform well on long timestamps. However, the algorithms including the pre-filtering operation have been applied to various combinations of value-equivalent tuple percentages and tuple lifespan distributions, producing similar results. For the remaining system characteristics and cost metrics, we used the same values used in [4] and [5], and set the hash computation cost to 4$\mu$sec. System characteristics and cost metrics are summarised in figures 3 and 4.

| **Parameter** | **Value** |
|---|---|
| Relation size | 16MB |
| Tuple size | 16 bytes |
| Tuples per relation | 1M |
| Timestamp size | 8 bytes |
| Explicit attribute size | 8 bytes |
| Relation lifespan | 100000 chronons |
| Page size | 1 K |
| Cluster size | 32 K |

**Figure 3 – System characteristics**

The pre-filtering predicate selectivity factors appearing in diagrams are 1 (all tuples qualify), 0.95, 0.85 and

0.75. We chose relatively high selectivity factors, since for small selectivity factors the performance advantage of the pre-filtering technique is obvious.

| Parameter | Value |
|---|---|
| Sequential I/O cost | 5 msec |
| Random I/O cost | 25 msec |
| Explicit attribute compare | 2 $\mu$sec |
| Timestamp compare | 4 $\mu$sec |
| Pointer compare | 1 $\mu$sec |
| Pointer swap | 3 $\mu$sec |
| Tuple move | 4 $\mu$sec |
| Hash computation | 4 $\mu$sec |

**Figure 4 – Cost metrics**

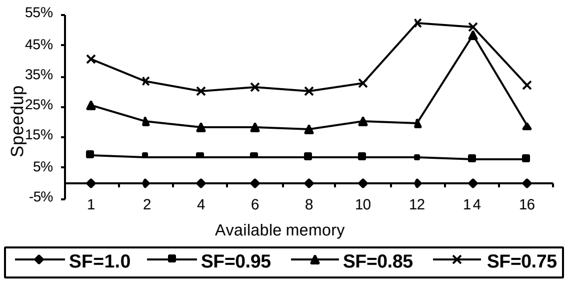

**Figure 5 – Speedup for the EP-H algorithm**

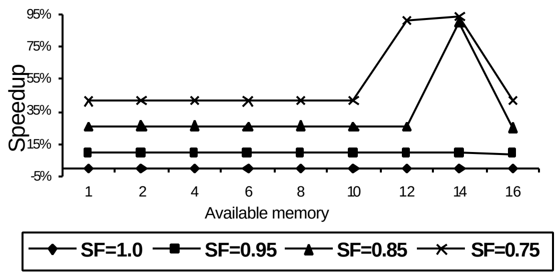

**Figure 6 – Speedup for the ES algorithm**

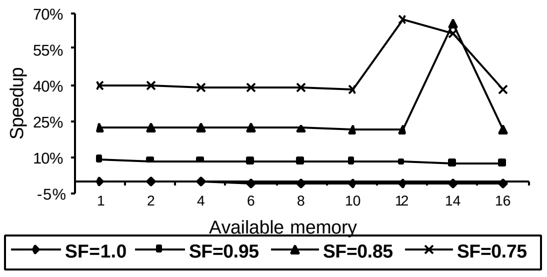

**Figure 7 – Speedup for the TP algorithm**

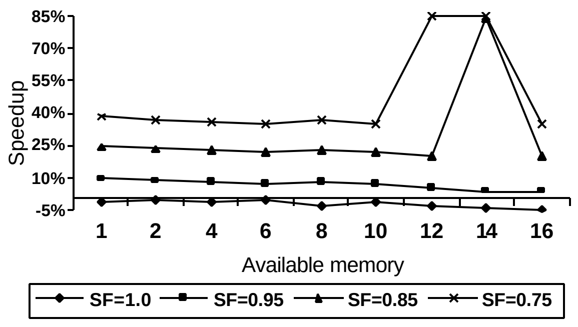

**Figure 8 – Speedup for the TS-H algorithm**

From the diagrams presented in figures 5-8 we can derive the following:

1. Even when the selectivity factor of the pre-filtering predicate is 1 and thus all tuples qualify, the performance penalty paid (due to the extra timestamp comparisons) is very small (an average of 0.7% with a worst case of 3.9%). Hence, an optimiser may employ pre-filtering "blindly", i.e. without knowledge about the selectivity factor of the predicate. Such knowledge would be useful, however, for buffer allocation decisions and, together with coalescing factor estimates, for accurate estimations of the size of the resulting relations.
2. In all diagrams we notice that the speedup obtained because of the introduction of pre-filtering increases considerably at some point of the x-axis, while it drops at a later point. These two points mark an area within which pre-filtering makes the difference between in-memory coalescing and coalescing that requires intermediate disk runs (or disk buckets). The location of the area starting point on the x-axis depends on the selectivity factor of the pre-filtering predicate.
3. For the remaining cases, i.e., when the pre-filtering predicate selectivity factor is less than 1 and both the optimised and non-optimised algorithms follow the same pattern of work (i.e., either they both operate completely in memory or they both write intermediate results on the disk), the average cost reduction obtained is higher than the percentage of rejected tuples. This is expected, since the asymptotic complexity of the coalescing operator is *O(N * log(N))* (by analogy to sorting and partitioning operators), thus decreasing *N* by a factor *r* should lead to a reduction in the overall cost that is higher than *r*. Moreover, for a given selectivity factor the speedup obtained is higher for small (available memory / relation size) ratios and decreases when more memory is made available (and thus in-memory operations become the dominant factor of the overall cost).

## 4 Conclusions

In this paper, we have presented an optimisation scheme which may be used to reduce the overall execution of time of coalesce/valid time selection operation sequences. This scheme consists of applying a pre-filtering predicate to the relation before coalescing, so as to eliminate tuples that may not contribute to the final result. The benefits resulting from employing this scheme have been verified via experiments using the Time-It testbed and illustrated diagrammatically. Implementing the proposed optimisation scheme has proved quite easy, by simply rejecting tuples not satisfying the pre-filtering predicate as soon as the corresponding page is read into memory. Future work will focus on identifying optimisable query execution plan patterns involving the coalesce operator and investigating cases that tailored coalescing algorithms may be employed.

**Acknowledgements:** We would like to thank Dr. M. Soo for making the latest versions of the Time-It testbed available, and Prof. Y. Ioannidis for his valuable comments.

## 5 References

[1] C. Ang and K. Tan, "The interval B-tree", Information Processing Letters, (52)2, pp.85-89, 1995.

[2] G. Ariav, "A Temporally Oriented Data Model", ACM Transactions on Database Systems 11(4), pp. 499-527, December 1986.

[3] N. Beckmann, et al., "The R$^*$–tree: An Efficient and Robust Access Method for Points and Rectangles", Proceedings of the 1990 ACM SIGMOD International Conference on Data Management, pp. 322-331, Atlantic City, N.J., June 1990.

[4] M. Böhlen, R. Snodgrass and M. Soo, "Coalescing in Temporal Databases", Proceedings of the 22nd VLDB Conference, Bombay, India, 1996.

[5] M. Böhlen, R. Snodgrass and M. Soo, "Coalescing in Temporal Databases", TimeCenter Technical Report TR-9, April 1997.

[7] J. Clifford and A. Croker, "The Historical Relational Data Model (HDRM) Revisited" in "Temporal Databases: Theory, Design and Implementation", edited by A. Tansel et al., Benjamin/Cummings Publishing Co., 1993.

[8] R. Elmasri, G. Wuu and V. Kouramajian, "The Time Index and the Monotonic B$^+$–tree", in "Temporal Databases: Theory, Design and Implementation", edited by A. Tansel et al., Benjamin/Cummings Publishing Co., 1993.

[9] R. Elmasri and S. Navathe, "Fundamentals of Database Systems", Addison-Wesley Publishing Company, 1994.

[10] Shashi Gadia, "Ben-Zvi's Pioneering Work in Relational Temporal Databases", in "Temporal Databases: Theory, Design and Implementation", edited by A. Tansel et al., Benjamin/Cummings Publishing Co., 1993.

[11] G. Graefe, "Query Evaluation Techniques for Large Databases", ACM Computing Surveys 25(2), pp. 73-170, June 1993.

[12] A. Guttmann, "R–trees: A Dynamic Index Structure for Spatial Searching", Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, pp. 47-57, June 1984.

[13] C. Jensen et al., "A Consensus Glossary of Temporal Database Concepts", available through http://www.cs.auc.dk/~csj/Glossary/. An older version was published in the ACM SIGMOD Record, 23(1), pp.52-64.

[14] N. Kline and M. Soo, "Time-It: The Time Integrated Testbed" Pre-Beta version 0.1, available via anonymous ftp through ftp://ftp.cs.arizona.edu/timecenter/time-it-0.1.tar.gz

[15] V. Kouramajian et al. "The Time Index$^+$: An Incremental Access Structure for Temporal Databases", Proceedings of the Third International Conference on Information and Knowledge Management (CIKM '94), pp. 296-303, Gaithersburg, Maryland, 1994.

[16] N. Lorentzos, "The Interval-extended Relational Model and Its Application to Valid Time Databases", in "Temporal Databases: Theory, Design and Implementation", edited by A. Tansel et al., Benjamin/Cummings Publishing Co., 1993.

[17] N. Lorentzos and Y. Mitsopoulos, "SQL Extension for Interval Data", IEEE TKDE, 9(3), pp. 480-499, 1997.

[18] M. Nascimento and M. Dunham, "Indexing Valid Time Databases via B$^+$-trees–The MAP21 Approach", TimeCenter Technical Report TR-26, March 1998.

[19] S. Navathe and R. Ahmed, "Temporal Extensions to the Relational Model and SQL", in "Temporal Databases: Theory, Design and Implementation", edited by A. Tansel et al., Benjamin/Cummings Publishing Co., 1993.

[20] B. Salzberg and V. Tsotras, "A Comparison of Access Methods for Temporal Data", TimeCenter Technical Report TR-18, June 1997.

[21] T. Sellis, N. Roussopoulos and C. Faloutsos, "The R$^+$–tree: A Dynamic Index for Multidimensional Objects", Proceedings of the Conference on Very Large Databases, pp. 507-518, Brighton, England, September 1987.

[22] H. Shen, B. Ooi and H. Lu, "The TP–Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases", Proceedings of the 10th IEEE International Conference on Data Engineering, pp. 274-281, Houston, Texas, February 1994.

[23] R. Snodgrass, "The Temporal Query Language TQuel", ACM Transactions on Database Systems, 12(2), pp. 247-298, June 1987.

[24] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass, "Temporal Databases: Theory, Design and Implementation", Benjamin/Cummings Publishing Company, 1993.

[25] The TSQL2 Language Design Committee, "The TSQL2 Temporal Query Language", edited by R. Snodgrass, Kluwer Academic Publishers, 1995.