

INCREMENTAL COMPUTATION METHODS
IN VALID & TRANSACTION TIME DATABASES

A Thesis
Presented to
The Academic Faculty

by

Mario Aleksic

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

Georgia Institute of Technology
December 1996

INCREMENTAL COMPUTATION METHODS
IN VALID & TRANSACTION TIME DATABASES

Approved:

Leo Mark, Chairman

Edward Omiecinski

Karsten Schwan

Date Approved _____

TABLE OF CONTENTS

LIST OF TABLES	vi
LIST OF FIGURES	vii
SUMMARY	viii
CHAPTER I INTRODUCTION	1
CHAPTER II CONCEPTS FOR THE STORAGE OF OBJECT-HISTORIES	5
Naming Conventions	5
The Item Relation	7
CHAPTER III THE IMPLICIT INVALIDATION MODEL	9
A backlog Model with Implicit Invalidation of Temporal Facts	10
Why Incremental Timeslice Computation Fails	12
CHAPTER IV THE EXPLICIT INVALIDATION MODEL	15
The Backlog	15
The Backlog Indices	16
Mapping Backlog Tuples to Bitemporal Elements	18
Validity Rectangles	18
Partially Unbounded Rectangles	20
CHAPTER V DIFFERENTIAL TIMESLICE COMPUTATION USING THE BACKLOG	22
Basic Database Operations	22
Insert	22
Delete	23
Update	25

Incremental Computations on Bitemporal Databases	26
Moving Through the Space of Validity Rectangles	27
Decremental Valid-Time Computation	29
Incremental Valid-Time Computation	32
Differential Transaction-Time Computation	33
Differential Computation Revisited	34
 CHAPTER VI	
STORAGE STRUCTURES FOR BACKLOG-INDICES AND TIMESLICES	38
 Transaction-Time Index	38
Valid-Time Index	40
An Alternative Storage Model for $BT_{R, VTIME, VTYPE}$	41
Storage Models for Timeslices	43
 CHAPTER VII	
TIMESLICES FOR TRANSACTION AND VALID-TIME PERIODS	45
 Extending Timeslice Relations	45
Defining Differential Computation for History Timeslices	47
Extending Differential Computation	48
Extending History Timeslices	49
Shrinking History Timeslices	51
 CHAPTER VIII	
COMPLEXITY OF THE ALGORITHM	52
 Update Processing	52
Query Processing	53
 CHAPTER IX	
COMPARISON OF THE PROPOSED ALGORITHM WITH EXISTING SOLUTIONS	57
 M-IVTT versus Explicit Invalidation Model	57
Bitemporal Interval Tree versus Explicit Invalidation Model	58
Comparing bitemporal timeslice algorithms – an example	60
Run-time of queries	61
 CHAPTER X	
EXTENDING THE ALGORITHM TO N-DIMENSIONAL SPACE	64
 Justification of Higher Dimensions of Time – an Example	64
3-dimensional Differential Computation	65
Generalization to n Dimensions	66

CHAPTER XI	
CONCLUSIONS	68
APPENDIX A	
PSEUDO CODE FOR DIFFERENTIAL BITEMPORAL TIMESLICE COMPUTATION	70
Incremental Transaction-Time Computation	70
Incremental Valid-Time Computation	71
Decremental Transaction-Time Computation	72
Decremental Valid-Time Computation	73
REFERENCES	74
GLOSSARY	76

LIST OF TABLES

Table

1	Attributes of the Item Relation T_R	7
2	Attributes of the Implicit Invalidation Backlog	10
3	Attributes of the Explicit Invalidation Backlog B_R	16
4	Attributes of History Timeslices $TS_R(ttbegin\ to\ ttend,\ vtime)$ and $TS_R(ttime,\ vtbegin\ to\ vtend)$	46
5	Summary of Existing Bitemporal Timeslice Algorithms	60
6	Sample Relation emp_proj	61
7	Runtimes of a Pure-Timeslice Query on the Sample Relation	63

LIST OF FIGURES

Figure

1	Validity Regions of the Sample Transactions	12
2	Bitemporal Element of the (Object, Item) Pair (Jack, Shipping)	20
3	Before the Delete Operation	23
4	After the Delete Operation	24
5	Validity Rectangle Bounded by Four Corners	28
6	Decremental Valid-Time Computation Step 1: Leaving Validity Rectangles at vtime 6	31
7	Decremental Valid-Time Computation Step 2: Entering Validity Rectangles at vtime 6	31
8	Decremental Valid-Time Computation Step 1: Leaving Validity Rectangles at vtime 6	36
9	Nested Access Structure for Valid Time	41
10	Decrementing Transaction Time for $TS_R(\text{ttime}, \text{vtstart to vtend})$ Step 2: Entering Validity Rectangles	49
11	Extending $TS_R(\text{ttbegin to ttend}, \text{vtime})$	50
12	A Validity Cuboid	65

SUMMARY

The topic of this thesis is the efficient implementation of the bitemporal timeslice query. It answers queries of the type: What knowledge about the state of the world at valid time, *vtime*, was current in the database at transaction time, *ttime*? Differential timeslice computation takes an existing cached timeslice as the outset and applies necessary changes to get the result of the new timeslice query. This is efficient because two timeslices with similar valid time and transaction time can be expected to have a large number of tuples in common. Differential computation has previously been applied to transaction-time databases. This scheme is extended to suit bitemporal databases. In addition, the algorithms are generalized for history timeslices that cover a transaction or valid time period, and to n-dimensional timeslices for applications with higher dimensions of time. Some aspects of physical data storage in an implementation of the algorithm are discussed, the complexity is analyzed and compared with existing solutions. Directions for future research are given.

CHAPTER I

INTRODUCTION

“A temporal database supports some aspect of time, not counting user-defined time [10].” More specifically, there are two orthogonal aspects of time that have been the focus of temporal database research: valid time and transaction time. Valid time is the time when something happens in the real world whereas transaction time is the time when something is recorded in the database. Ideally, we would like to store everything into our temporal database as soon as it happens, and the data we store would always be correct. In such an ideal scenario, the valid time and transaction time are the same and there is no need to distinguish between them. There are cases where reality is what is stored in the database. Bank accounts are a typical example for this. Whenever money is deposited into the account or withdrawn from it, the new balance is stored in the database. The stored account balance always reflects reality. If a deposit is not stored, money actually gets lost, if a withdrawal is not stored, money is created. Such systems are called transaction-time databases. Timestamps are generated automatically by an internal clock. Naturally, transaction timestamps are monotonically increasing.

In a valid-time database, it is only of interest when something happened in reality. This is unrelated to the time when it is stored. Therefore, valid timestamps have to be supplied by the user. Valid timestamps can be entered in any order, in case of an error, they can be deleted or changed.

A bitemporal database combines user-supplied valid time with system-generated transaction time. Every time the user enters data, the database acquires knowledge about facts that are supposedly true in the real world during some period of valid time. This knowledge about the real world changes over time as new facts are added and existing facts are found to be false. A bitemporal database keeps track of the knowledge about the real world that changes as the internal clock (transaction time) advances.

While the ideas of temporal databases are known since the introduction of bookkeeping, computerized models have been proposed a few decades ago. The first transaction-time model was published by Schueler in 1977 [19]. The first bitemporal model was presented by Ben-Zvi in his Ph.D.-dissertation in 1982 [4]. For a historical overview of temporal database research, see [20]. However, practically useful temporal database systems were not feasible until recently when the storage of massive amounts of data became affordable. Nevertheless, the increase in storage capacity is not accompanied by a comparable increase in access speed. Modern disks show an acceptable bandwidth when it comes to transferring large amounts of consecutive disk blocks, but the time necessary to find a random disk block improves only slowly. Therefore, in most applications not space but access speed is the bottleneck today. As the gap between storage capacity and access speed widens, this will be true for a growing number of applications in the foreseeable future.

The standard query that a temporal database must support is the timeslice query. It requests the state of the data at some point in transaction and/or valid time. For bitemporal relations, this query would be: What knowledge was current in the database at transaction time *ttime* about the state of the data at valid time *vtime*?

Incremental computation was first used as a lazy evaluation scheme that allows fast insertions into traditional databases by storing update requests in a backlog and updating the base relations when the next query is asked [18]. The first approach to generalize this scheme to increment and decrement timeslices for transaction-time relations was [11], in [17], differential (incremental and decremental) computation was used to update valid time trees for bitemporal relations. [1] introduced a differential scheme for bitemporal relations. This work is a logical extension to the scheme presented in [11], major parts of it appear in this thesis.

Besides incremental methods, special tree-structures for accessing bitemporal relations have been used. One possibility is the use of R-trees [8] which were originally designed for spatial data. The bitemporal interval tree (BIT) was specially designed for bitemporal relations [15].

Besides describing the differential computation method introduced in [1], this thesis gives a more detailed description of the underlying physical data structures, provides a thorough analysis of the complexity and extends the algorithm to more general types of timeslices such as history timeslices and n-dimensional timeslices. Finally, the algorithm is compared with the two existing indexing methods for bitemporal relations: [15] and [17].

The rest of the thesis is organized as follows: CHAPTER II introduces some basic terms and concepts used throughout the thesis, CHAPTER III discusses an ad-hoc solution for bitemporal timeslices which is too inefficient to be useful, and CHAPTER IV presents the explicit invalidation model which is the used for differential computations as described in CHAPTER V. CHAPTER VI gives the physical storage structures, CHAPTER VII extends standard bitemporal timeslices to history timeslices which cor-

respond to a period in transaction or valid time rather than a single instant. The complexity of the algorithm is analyzed in CHAPTER VIII, CHAPTER IX compares the performance with the Multiple Incremental Valid Time Tree [17], and the Bitemporal Interval Tree [15]. Finally, CHAPTER X extends the algorithm to n-dimensional space and CHAPTER XI concludes the thesis.

CHAPTER II

CONCEPTS FOR THE STORAGE OF OBJECT-HISTORIES

Naming Conventions

In relations of conventional relational database systems (snapshot relations), each tuple represents an object such as an entity with a counterpart in the real world or an instance of a relationship. I only consider relations in first normal form, therefore there is only one value stored in the database for each attribute of a tuple. In order to capture the time-varying nature of data, there are two possible extensions to the relational model: tuple timestamping and attribute timestamping [21]. Attribute timestamping abandons the first normal form and associates multiple values and timestamps with each attribute of a tuple. In this approach, a tuple's attribute values are a function of time. With tuple timestamping, a new tuple is generated every time an attribute value changes and timestamps are associated with the whole tuple. The approach used in a specific data model should be treated as an implementational detail of that model and it should not be visible to the user. TSQL2 provides a conceptual model capturing the semantics of temporal relations without being restricted to a specific internal representation [14]. Instead of referring to time-varying attribute values or timestamped tuples, [14] more generally refers to "facts" stored in temporal relations. Translating back to tuple and attribute timestamping, a "fact" can be that a set of attribute values, or a whole tuple is valid during some period or at a point in time.

In CHAPTER III and CHAPTER IV, I will introduce two data models for bitemporal relations, both of them use tuple timestamping. The tuples and their corresponding timestamps will be kept in two separate relations in first normal form. Entries in these relations are tuples themselves. Moreover, two tuples can be two different versions of the same thing and could be viewed as one tuple whose attribute values change over time. In fact, two such tuples would be one tuple if attribute timestamping was used. Depending on the point of view, “tuple” can have three different meanings. Instead of using “tuple” in this ambiguous way, I will use the following terms:

- “object” is used to describe any object being modeled from reality. An “object” corresponds to a tuple in the case where attribute timestamping is used.
- “item” describes a set of atomic attribute values. The pair (object, item) describes the “fact” that “object” has attribute values specified by “item”. (Object, item) can have multiple timestamps describing when the fact is valid and corresponds to the common use of “tuple” in a tuple timestamped environment.
- “tuple” is used in the original meaning describing an entry in a conventional relation. Throughout this thesis, “tuple” will be used when discussing physical data models for temporal relations. It always corresponds to physically stored data.

The use of “item” and “object” was inspired by [12] where objects are introduced as entities with an existence in the real world and items as single states of objects.

The models I will introduce in this thesis are “history-oriented” in the sense defined in [10] (see Glossary). In order to store object histories, a DBMS must support “history unique identification”, i.e. there must be a unique identifier for each object in the database in order to identify all items that belong to the same object. As such, object identifiers are time-invariant. I will use the surrogate data type for both object and item

identifiers. The cent-sign “¢” will be used as a suffix for all attribute-names that have this data type. For a good definition of surrogates, see [13]. There is one history-related integrity constraint supported by the proposed models: At any time instant, an object can have at most one item associated with it.

The following introduces the item relation which is used to store atomic attribute values (items) of a temporal relation. Major portions of it appeared in [1]. The two models proposed in CHAPTER III and CHAPTER V have the item relation in common. They differ in the structure of the backlog where all update activity is stored.

The Item Relation

The item relation, T_R , records the changes in state of individual objects in the database. The item surrogate (item¢) is the primary key of the item relation and is used as a foreign key in the backlog. Attributes $A_1 \dots A_n$ are the time-varying attributes of objects in R (see Table 1). When an insert or update transaction is performed on R , a new item surrogate is generated and its attributes are stored in T_R . When a delete transaction for an item is entered into the backlog, the item surrogate will identify the relevant tuple already present in the item relation. There is no need to record the attributes of the item again.

Table 1: Attributes of the Item Relation T_R

attribute	domain
item-surrogate	surrogate
A_1	$\text{dom}(A_1)$
A_2	$\text{dom}(A_2)$
A_3	$\text{dom}(A_3)$
...	...

As opposed to the method used in [11] where items are stored in the backlog, we decided to store them separately. There are several advantages for this: First, it reduces the size of the database because the attribute data for an individual item is recorded only once when the item is inserted into the database. In [11], attribute data is stored twice: when the item is inserted and when it is logically deleted. The second advantage stems from the difference in differential timeslice computation between transaction time and bitemporal relations. As described in [11], every backlog tuple read in differential computation for transaction-time relations either results in an insertion or deletion of a tuple in the timeslice. As we will see in CHAPTER V, not all backlog tuples read in differential computation for bitemporal relations result in a manipulation of the timeslice. Therefore the percentage of backlog tuples whose corresponding item is not needed is much higher and it is a big advantage that the unnecessary items are not read in the pass through the backlog. During differential computation, we collect the relevant item surrogates in a pass through the backlog. We then obtain the object states from the item relation. Once we compute the target timeslice, we can minimize the time to retrieve the tuples from the item relation by pulling all tuples located on the same page at one time.

CHAPTER III

THE IMPLICIT INVALIDATION MODEL¹

Two methods exist for invalidation of temporal facts, explicit and implicit. With explicit invalidation of temporal facts, we insert an extra tuple into the backlog when we pair a new item with an object. This tuple contains the value of the last item (in transaction time) associated with the object. When we use the implicit method of invalidation, we only insert a tuple with the new value of the object. We do not include a tuple with the older value. When we run across the new value of the object during timeslice computation, we then know the old value is invalid. With explicit invalidation of temporal facts, we have a larger backlog because we must store the additional invalidation tuples. Also, we have more expensive insert and update operations because we must obtain the old values of the object before placing the new item tuple in the backlog. However, we will demonstrate that timeslice computation during queries is cheaper than with implicit invalidation. With implicit invalidation of temporal facts, the backlog is smaller, because we do not store the invalidation tuples. Also, the insert and update operations are cheap, because we simply add the new information without looking up the older values of the object. Despite these clear advantages over explicit invalidation, we will see that implicit invalidation of temporal facts makes querying the database expensive. We will show that implicit invalidation complicates timeslice computation. We will demonstrate that differential timeslice computation is not possible.

1. This chapter also appeared in [1].

A backlog Model with Implicit Invalidation of Temporal Facts

In the implicit invalidation model, temporal facts are stored directly into the backlog. The backlog structure suggested here is essentially the same as Jensen's backlog-based scheme presented in [14] except that the type of operation is not stored (we always assume inserts, a delete has a NULL-value in $item\phi$) and $item\phi$ is used to identify attribute values stored in the item relation instead of storing the attribute values in the backlog. Table 2 shows the attributes of the backlog B_R used in the implicit invalidation model.

Table 2: Attributes of the Implicit Invalidation Backlog

attribute	domain
$object\phi$	surrogate
$item\phi$	surrogate
ttime	timestamp
vtstart	timestamp
vtend	timestamp

In the implicit invalidation model, a bitemporal timeslice $TS_R(tt, vt)$ is defined as the collection of (object, item) pairs which are valid at transaction time tt and valid time vt . For each object O the current item at valid time vt and transaction time tt can be retrieved from the backlog by selecting the tuple V with the highest transaction time among those tuples with $vtstart(V) \leq vt \leq vtend(V)$ and $ttime(V) \leq tt$. This expresses the fact that updates affecting a point in valid time vt and performed at transaction time tt invalidate all insertions and updates which affect the same valid time vt which were performed at transaction times earlier than tt .

Formally, the set of tuples in a timeslice $TS_R(tt, vt)$ can be described as:

$$TS_R(tt, vt) = \{(O, I) \mid \exists (r \in B_R) : r.object\phi = O \wedge r.item\phi = I \wedge \\ r.vtstart \leq vt \leq r.vtend \wedge \\ r.ttime(t) = \max\{ v.ttime \mid v.vtstart \leq vt \leq v.vtend \wedge \\ v.object\phi = O \}$$

Consider the following example to visualize the validity regions of temporal facts in the valid time/transaction time grid (Figure 1). A further analysis of the example will show that this is a case where differential computation fails. Transactions are labeled with increasing numbers as their transaction time. This type of graphical representation is taken from [14].

- transaction 1: Jack works for the shipping department from January and on
(region with light shading)
- transaction 2: Jack changes to the marketing department in March (2nd lightest region)
- transaction 3: Jack changes to the administration department in May (3rd lightest region)
- transaction 4: previous information was erroneous, Jack worked for the postal department between February and June (darkest region)

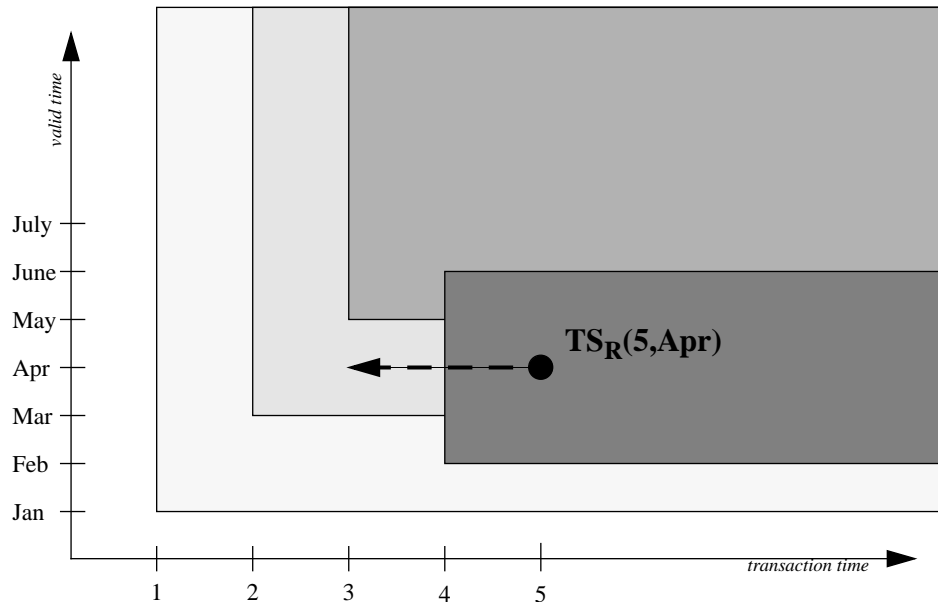


Figure 1: Validity Regions of the Sample Transactions

When an (object, item) pair is entered, its validity region in the grid is represented by a rectangle. The rectangle is bounded by the starting valid time, vt_{start} , the ending valid time, vt_{end} , the current transaction time tt , and transaction time NOW. It overrides previously existing validity regions. For each point (tt, vt) in Figure 1, the shading determines which attribute values are valid for the object “Jack”. For example, in $TS_R(5, Apr)$, the value for Jack’s department is “postal” as set in transaction 4.

Why Incremental Timeslice Computation Fails

During incremental and decremental timeslice computation with implicit invalidation of temporal facts, one selects an already computed and cached timeslice near the target point in time and appropriately applies transactions between the two points. In our example, we might want to compute the timeslice $TS_R(3, Apr)$ by using the existing

timeslice $TS_R(5, Apr)$ and undoing the only transaction between those two points: transaction 4. The arrow in Figure 1 depicts the direction of this decremental computation. In order to undo transaction 4, one must retrieve the earlier value of Jack's department which was current before transaction time 4. This is a relatively difficult task. The data stored about transaction 4 in the backlog does not contain this information. There can be different old values depending on the valid time one is interested in. In our example, old values of transaction 4 can be those set by transaction 1, 2 or 3. The temporal facts stored by these three transactions are the ones that may become invalid at transaction time 4. In general, the number of previous values can be very high.

A way out might be to create an index on the backlog that allows one to find the old values by performing a search on the index. One looks for tuples in the backlog that have the same object ϕ as the transaction to be undone and a starting and ending valid time covering the valid time of the timeslice. Among the tuples matching this description, we need the one with the highest time that is smaller than the one of the transaction to be undone. Unfortunately, none of the possible combinations of transaction and valid times of the transactions result in a suitable index as explained below.

Since only transactions on a single object are under consideration, object ϕ is the first field in the index. The second field would have to be one of the following timestamps.

- *ttime*: would allow us to select the transaction with the highest transaction time, but this object might not be valid at the valid time of the timeslice (transaction 3 in the example);
- *vtstart*: one could select the transaction that starts being valid at the latest time before the timeslice's valid time. This will not work because the transaction's

validity could still end before that time. There might be a temporal fact whose validity starts earlier but ends after the timeslice's valid time. This would be the one that has to be selected.

- vtend: here, the argument is similar to the one for vtstart; the selected transaction would cease to be valid shortly after the timeslice's valid time, but might start being valid after the timeslice's valid time as well. A temporal fact that covers the timeslice's valid time might have a later ending valid time.

The approach above may be both space-efficient and allow for quick storage of temporal facts. However, differential timeslice computation is not feasible.

CHAPTER IV

THE EXPLICIT INVALIDATION MODEL¹

The implicit invalidation model is a poor choice if one wants to perform differential computation. On the other hand, the explicit invalidation model allows both incremental and decremental computation. We propose a database structure that is only slightly different from the one suggested in CHAPTER III, but we use it in a different way. As discussed in CHAPTER II, we split the backlog into two relations, the backlog and the item relation. We record the transaction types and timestamps in the backlog. The item relation holds the state of an item's attributes when we enter it into the database. We link these two tables, as before, by the item surrogate. The item surrogate is the primary key for the item relation. When we perform an operation on the backlog, we append the new information to the tails of the backlog and item relation.

The Backlog

Each tuple in the backlog has a transaction type (ttype) of either “insert” or “delete” and a valid time type (vtype) of either “begin” or “end” (see Table 3). The item surrogate identifies the corresponding tuple in the item relation. The object surrogate identifies the modified object. For example, an object surrogate might be used to represent a specific employee. As the employee's salary changes, new item surrogates would be created for each change in salary but the object surrogate would remain unchanged. An insertion (INS) of a beginning valid time (BEGIN) represents the fact that at *ttime*

1. This chapter is also included in [1].

a given item is known to start existing at the given valid time $vtime$. The other combinations of $ttype$ and $vtype$ work similarly. A detailed example is given later in this chapter. An update of an object is recorded as a deletion of the outdated item followed by an insertion of the new item for that object[11]. We never physically remove data from the database.

The primary key of B_R is $(object\phi, ttime, vtime, ttype, vtype)$. $item\phi$ is the only attribute not included in the primary key. This is because there can be at most one item associated with an object at any point in the transaction time/valid time grid.

Table 3: Attributes of the Explicit Invalidation Backlog B_R

attribute	domain
$item\phi$	surrogate
$object\phi$	surrogate
$ttype$	{INS,DEL}
$vtype$	{BEGIN, END}
$ttime$	timestamp
$vtime$	timestamp

The Backlog Indices

The algorithms for differential timeslice computation described in CHAPTER V rely on a certain order of the backlog tuples. Differential transaction-time computation needs the tuples in the order of transaction-time index I_T whereas differential valid-time computation expects the tuples sorted according to valid-time index I_V .

Transaction-time index I_T on the backlog B_R (primary index, physical ordering):

1. ttime
2. object ϕ
3. ttype (where DEL < INS)
4. vtime
5. vtype (where END < BEGIN)

As transactions on the database naturally occur at distinct, increasing transaction times, the physical ttime-ordering is assured automatically. However, it is not assured that all the tuples belonging to one transaction are posted in the ordering of object ϕ , ttype, vtime and vtype. To fix this, there must be an intermediate process that first assembles all backlog tuples for one transaction and then stores them to the backlog according to index I_T .

Valid-time index I_V on the backlog B_R :

1. vtime
2. object ϕ
3. vtype (where END < BEGIN)
4. ttime
5. ttype (where DEL < INS)

It is the nature of valid time that timestamps can be entered into the database in any order. As opposed to the transaction-time index, it is therefore necessary to maintain an index structure for I_V that allows for out-of-order insertion. For the moment, we will assume a simple B^+ -tree for index I_V . Possible index structures are discussed in more detail in CHAPTER VI.

Mapping Backlog Tuples to Bitemporal Elements

In the conceptual bitemporal model presented in [14], *bitemporal chronons* are atomic instances in the valid time/transaction time grid. They do not necessarily have the same granularity for both time axes. For example, the granularity for transaction time could be a thousandth of a second and the granularity for valid time one day. An example for a chronon in this space is (1/1/1996, 5/1/1996 10:05:23,015 am).

An (object, item) pair is associated with a set of bitemporal chronons called its *bitemporal element* [14]. Recall that an item stores the state of an object at a particular instant in time and this state is valid within the region covered by its bitemporal element.

Validity Rectangles

In our model, bitemporal elements are not stored explicitly as a collection of bitemporal chronons. Instead, the bitemporal elements are partitioned into validity-rectangles, also called *bitemporal intervals* [10]. In some models, e.g. Snodgrass' Representation Scheme [14], validity rectangles are stored by including four timestamps into a tuple. In our model, we will store the coordinates of the four corners $C_1 \dots C_4$ of validity rectangles in separate backlog tuples instead. This is because tuples in the backlog never change. In Snodgrass' model, the ending transaction time of a validity rectangle cannot be set when the rectangle is inserted into the database because it is not known when the corresponding fact will be logically deleted. Next is a simple example of how such a rectangle maps to bitemporal chronons.

Suppose for simplicity that the granularity for valid time is one month and that transactions are timestamped by sequential numbers in order of their occurrence. At transaction time 2, we enter the fact that Jack works for the shipping department be-

tween March and May (Figure 2). This results in two corners, C_1 and C_2 , being stored in the backlog: $item\phi$, $object\phi$ and $ttime$ are the same for C_1 and C_2 because both belong to the same operation. The $ttype$ is INS for both tuples. The starting valid time for the (object, item) pair (Jack, shipping) is March, so $vtime(C_1) = \text{March}$ and $vtype(C_1) = \text{BEGIN}$. The ending valid time is June, so $vtime(C_2) = \text{June}$ and $vtype(C_2) = \text{END}$. This definition of ending valid time differs from Jensen's [14]. In Jensen's model, the ending valid time is May. The bitemporal element includes chronons with the ending valid time in Jensen's model. In our model they are excluded. We considered this choice more natural because the ending transaction time is excluded both in Jensen's and our model.

At transaction time 5, the mentioned (object, item) pair gets logically deleted. This results in the tuples C_3 and C_4 being inserted into the backlog with the same attribute values as C_1 and C_2 , respectively, besides $ttype$ which is DEL for both of them. The following graph visualizes the four transactions and the corresponding bitemporal element consisting of the chronons marked by dots:

Later discussions of this algorithm will draw validity regions as shaded rectangles and implicitly exclude the upper boundaries from the region. The example above would be drawn as a rectangle with the corners C_1, C_2, C_3 and C_4 . This kind of graphical representation is derived from the description of the TSQL2 data model [14]. Notice that the subscript of the corners represents the order in which they are physically stored in the backlog. This ordering is crucial for differential timeslice computation described in CHAPTER V.

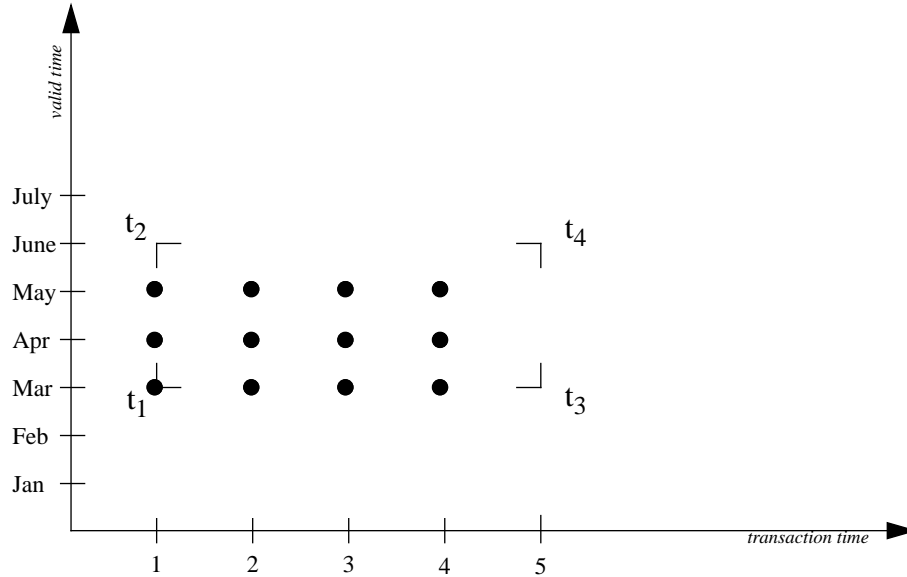


Figure 2: Bitemporal Element of the (Object, Item) Pair (Jack, Shipping)

Partially Unbounded Rectangles

There are many situations when one or more of the four corners of a validity region are unknown. In the example just discussed, the records C_3 and C_4 do not exist in the backlog prior to transaction time 5. At transaction times 1,2,3 and 4 the rectangle with corners C_1 and C_2 is unbounded in transaction time. In some bitemporal data models like Snodgrass' tuple timestamped representation scheme [14] this unboundedness is stored by means of a special timestamp called UC (until changed). In our model, as in Jensen's backlog-based scheme [14], the unboundedness is expressed by the absence of C_3 and C_4 . The only deviation from Jensen's model is the separate storage of C_3 and C_4 .

In addition to the unboundedness in positive transaction time, rectangles can be unbounded in positive valid time. Again, we express this unboundedness by the absence

of the corners that limit the rectangle in valid time. In our example, Jack works for the shipping department but there are no plans to dismiss him from that department. This fact would be stored by omitting C_2 and C_4 . Notice that the rectangle is unbounded in both transaction time and valid time at transaction times 1,2,3 and 4. The following algorithms depend on the fact that rectangles can only be unbounded with respect to ending valid or transaction time. Unboundedness with respect to beginning valid or transaction time is not permitted.

CHAPTER V

DIFFERENTIAL TIMESLICE COMPUTATION USING THE BACKLOG

Basic Database Operations

There are three basic operations on the database: insert, delete, and update. To simplify the use of temporal relations in application programs, implementation details such as the structure of B_R and T_R can be hidden. Instead, the application layer can present a database with extended update and query capabilities. Insertions, deletions, and updates on the database are extended by two valid timestamps specifying the beginning and ending valid time of the operation, respectively. As we allow unboundedness in valid time, the ending valid time can have the special value “Forever” [10] in our procedure calls. The transaction timestamp is implicitly provided by an internal clock.

Insert

The simplest operation is insert. Because insert generates a new object, we do not have to worry about validity rectangles for the same object that could be influenced by the transaction. The notation is derived from the discussion of the TSQL2 data model in [14].

```
procedure insert(R, vtstart, vtend, “tuple”);; insert new object “tuple” into R valid from vtstart to vtend
  objID = new-object-surrogate()
  itemID = new-item-surrogate()
  insert  $T_R$ (itemID, “tuple”) ;; store tuple-values as a new item
  insert  $B_R$ (itemID, objID, INS, BEGIN, NOW, vtstart)
  if vtend  $\neq$  Forever then
    insert  $B_R$ (itemID, objID, INS, END, NOW, vtend)
  fi
end
```

Notice that each transaction is atomic as far as transaction timestamps are concerned. This means that the value “NOW” has the same value both times it is used in the insert procedure. Both update and delete transaction may involve multiple entries to be stored in the backlog, all timestamped with the same value of NOW.

Delete

The next fundamental operation is the delete operation. It is used as a subroutine for the update operation as well. The delete procedure can be best explained by the graphs in Figure 3 and Figure 4:

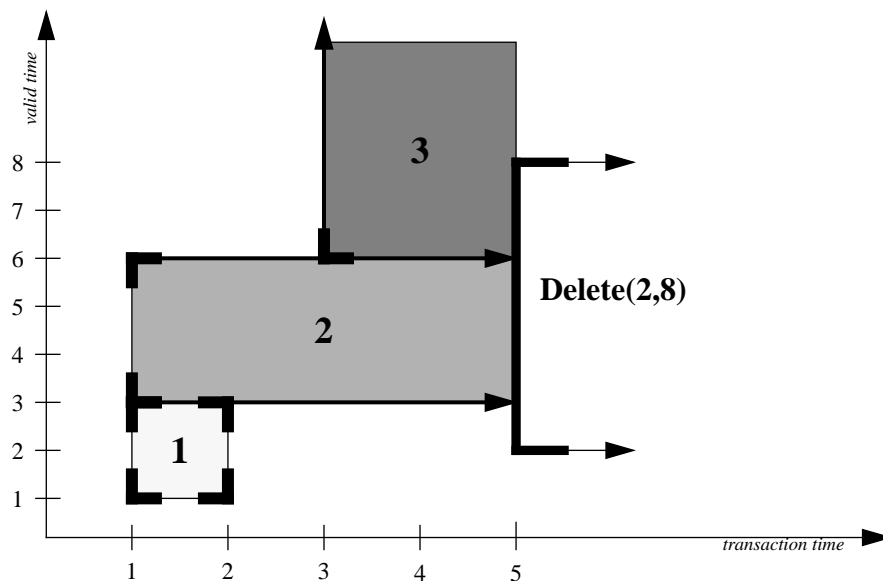


Figure 3: Before the Delete Operation

The graph above shows three different items and their temporal elements for some object O in a bitemporal relation R. At transaction time 5, the operation $\text{delete}(R, O, 2, 8)$ deletes object O between valid time 2 and 8. At transaction time 5 only the va-

validity regions of item 2 and 3 are unbounded in transaction time. Item 1 has already been deleted at transaction time 2 so its validity region is not affected by the newly entered delete. Item 2's valid times are covered by the valid times of the deletion and it therefore has to be logically deleted. Item 3's valid times are partially covered by the deletion and its validity region must only be partially bounded. This results in the graph of Figure 4:

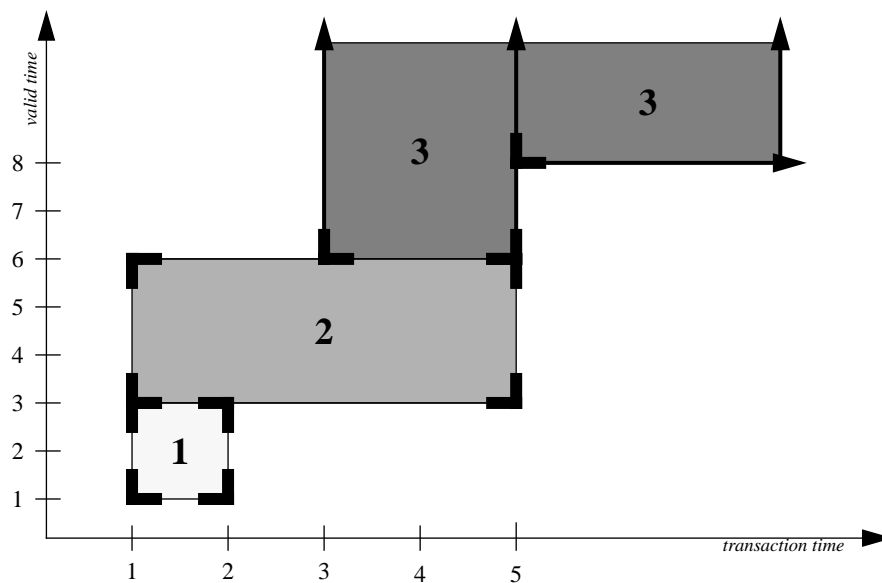


Figure 4: After the Delete Operation

The rectangles for item 2 and item 3 have been bounded at transaction time 5. Because the rectangle for item 3 was only partially covered, a new rectangle for the same item is inserted. It is bounded by valid time 8 and transaction time 5.

This deletion results in four entries being stored in the backlog, each entry corresponding to one corner of a validity rectangle inserted at transaction time 5. Two cor-

ners for item 2, one corner to limit the validity of the original rectangle for item 3, and one corner to establish a second rectangle for item 3 (see Figure 4).

In order to have quick access to all validity rectangles that are current at transaction time NOW, we will use the transaction-time timeslice for transaction time NOW. This timeslice returns the valid-time history for the specified transaction time and it is a special case of a history timeslice which is to be defined in CHAPTER VII: $TS_R(ttime, 0 \text{ to Forever})$. The structure of the timeslice relation is defined in Table 4 in CHAPTER VII.

```

procedure delete(R, objID, vtstart, vtend)  ;; delete object objID from relation R
                                           ;; between valid time vtstart and vtend
for all t  $\in$   $TS_R(\text{NOW}, 0 \text{ to Forever})$ : object $\phi(t) = \text{objID}$  and
                                           begin(t) < vtend and end(t) > vtstart           ;; overlaps with (vtstart->vtend)
  DO
    insert  $B_R(\text{item}\phi(t), \text{objID}, \text{DEL}, \text{BEGIN}, \text{NOW}, \text{begin}(t))$            ;; close current rectangle
    if end(t)  $\neq$  Forever then
      insert  $B_R(\text{item}\phi(t), \text{objID}, \text{DEL}, \text{END}, \text{NOW}, \text{end}(t))$ 
    fi

    if end(t) > vtend then                                           ;; existing rectangle is
      insert  $B_R(\text{item}\phi(t), \text{objID}, \text{INS}, \text{BEGIN}, \text{NOW}, \text{vtend})$            ;; still valid after vtend:
      if end(t)  $\neq$  Forever then
        insert  $B_R(\text{item}\phi(t), \text{objID}, \text{INS}, \text{END}, \text{NOW}, \text{end}(t))$            ;; insert new rectangle
      fi
    fi

    if begin(t) < vtstart then                                           ;; existing rectangle starts
      insert  $B_R(\text{item}\phi(t), \text{objID}, \text{INS}, \text{BEGIN}, \text{NOW}, \text{begin}(t))$            ;; being valid before vtstart
      insert  $B_R(\text{item}\phi(t), \text{objID}, \text{INS}, \text{END}, \text{NOW}, \text{vtstart})$            ;; insert new rectangle
    fi
  OD
end

```

Update

The implementation of the update operation is straightforward. Like in the insert operation we have to generate a new rectangle in the backlog that spans the validity region of the new item. Before we can do this, however, we have to ensure that old infor-

mation that might be stored in the database about the same valid-time period is logically deleted. This is done by simply calling the delete operation we just described:

```

procedure update(R, objID, vtstart, vtend, "tuple"); set the values of object objID to "tuple" in the given
    ;; valid time period (vtstart->vtend)
    delete(R, objID, vtstart, vtend)
    itemID = new-item-surrogate()
    insert TR(itemID, "tuple") ;; store tuple-values as a new item
    insert BR(itemID, objID, INS, BEGIN, NOW, vtstart)
    if vtend ≠ UC then
        insert BR(itemID, objID, INS, END, NOW, vtend)
    fi
end

```

Incremental Computations on Bitemporal Databases

In [11], a method for optimizing incremental computation in transaction-time databases was presented. In this method a timeslice for a given transaction time is cached as a set of pointers into the backlog. To compute the state of the database at some desired transaction time, one starts at a nearby cached timeslice and follows the backlog incrementally or decrementally, calculating the state of the database at the desired transaction time. We use a similar method for calculating bitemporal timeslices. The result of the bitemporal timeslice operator $TS_R(ttime, vtime)$ is a snapshot relation that contains the state of the objects in R as known at transaction time, $ttime$, which are valid at valid time, $vtime$. Notice that there is no constraint on valid time – it can be in the past as well as in the future, whereas the transaction time must be less than or equal to NOW.

Incremental and decremental timeslice computation takes an existing timeslice $TS_R(tt_0, vt_0)$ and performs all necessary transactions to obtain the desired timeslice $TS_R(tt_D, vt_D)$. The amount of necessary changes to the timeslice is relatively small if the desired timeslice and the given timeslice are “close” to each other in the transaction time/valid time space. It is not a topic of this paper how optimal outsets for differential

computations are found. For transaction-time databases, this has been solved by using the Pointer-less Insertion Tree (PLI-tree) [22]. This method can be used for differential transaction-time computation in bitemporal databases, too. For differential valid-time computation, further research is needed.

In addition to physically stored timeslices there is also a set of empty timeslices that can be used as a starting point for incremental computation: $TS_R(tt, 0)$ and $TS_R(0, vt)$ are empty for all tt and vt . The first describes timeslices for valid time equal to zero, that is, the beginning valid time. In [10], *Beginning* is defined as “a special valid-time instant preceding the smallest chronon on the valid-time line”. All items must be inserted into the database after this valid time. Depending on the used valid-time calendar this might be the beginning of the calendar system or some other useful limit like the time when a company shipping operations began. Similarly, transaction time zero is the time when the relation was created. At that time, the database was empty. This transaction time is called *Initiation* in [10]. For simplicity, the value “0” will be used for both transaction and valid time instead of *Beginning* and *Initiation*.

Moving Through the Space of Validity Rectangles

The location of a given timeslice $TS_R(tt_0, vt_0)$ in the valid time/transaction time grid is the bitemporal chronon with coordinates (tt_0, vt_0) . This timeslice is correct with respect to all validity rectangles in the grid. To compute the desired timeslice $TS_R(tt_D, vt_D)$, we will move through the graph in two steps: along the transaction-time axis in one step and along the valid-time axis in the other. The order these two steps are performed in does not matter as far as the result is concerned. In general, one of the orderings may be more efficient, but it is impossible to decide which ordering is better

without extensive knowledge about the structure of the stored transactions in that area of the database.

Whether a bitemporal chronon (tt, vt) is inside some validity rectangle can be determined from the four corners of the rectangle $C_1C_2C_3C_4$. Each corner of the validity region bounds the region in two directions. The corner is said to span the region which it bounds. For example if C_2 is (TT, VT) then C_2 spans the region $tt \geq TT$ and $vt < VT$. (tt, vt) is in the validity rectangle if and only if it is in the intersection of the regions spanned by $C_1, C_2, C_3,$ and C_4 . $TS_R(tt, vt)$ can be shown to be in the validity rectangle $C_1C_2C_3C_4$. Note that $C_2, C_3,$ and C_4 may not be in the backlog if the validity region is unbounded. Only C_1 is guaranteed to exist for every item.

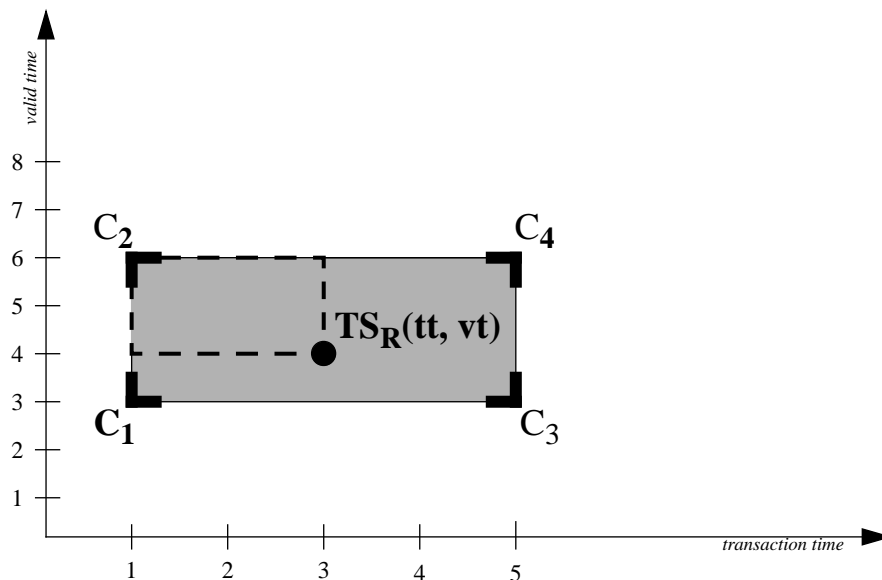


Figure 5: Validity Rectangle Bounded by Four Corners

C_1 through C_4 correspond to tuples in the backlog B_R with the following properties: for each individual validity rectangle:

- $\text{object}\phi(C_1) = \text{object}\phi(C_2) = \text{object}\phi(C_3) = \text{object}\phi(C_4)$,
- $\text{item}\phi(C_1) = \text{item}\phi(C_2) = \text{item}\phi(C_3) = \text{item}\phi(C_4)$,
- $\text{ttype}(C_1) = \text{ttype}(C_2) = \text{INS}$, $\text{ttype}(C_3) = \text{ttype}(C_4) = \text{DEL}$,
- $\text{vtype}(C_1) = \text{vtype}(C_4) = \text{BEGIN}$, $\text{vtype}(C_2) = \text{vtype}(C_3) = \text{END}$.

The fact that $\text{TS}_R(\text{tt}, \text{vt})$ is in the region spanned by C_2 can be demonstrated as follows: $\text{time}(C_2) \leq \text{tt}$ and $\text{vtime}(C_2) > \text{vt}$. Recall from the discussion of the granularity of time that lower bounds belong to the region and upper bounds do not. C_2 is a lower bound for transaction time and an upper bound for valid time. The regions spanned by C_1 , C_3 and C_4 can be defined similarly, and it can be shown that $\text{TS}_R(\text{tt}, \text{vt})$ is contained the validity regions of C_1 , C_3 , C_4 . Since $\text{TS}_R(\text{tt}, \text{vt})$ is in each of the validity regions, it must also be in their intersection, so $\text{TS}_R(\text{tt}, \text{vt})$ must be in the validity rectangle $C_1C_2C_3C_4$.

Decremental Valid-Time Computation

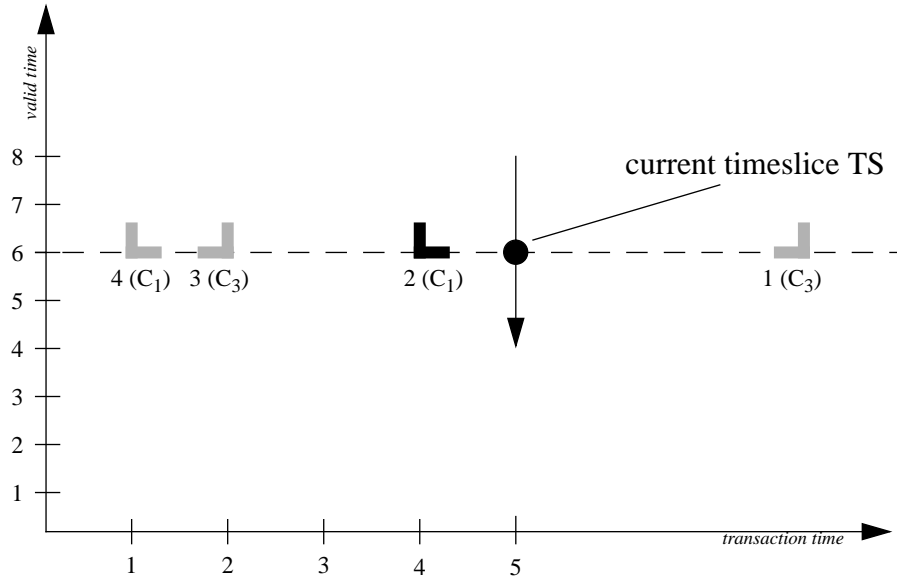
Figures 6 and 7 explain how decremental valid-time computation works. The two graphs show the tuples stored in the backlog for some object O_i at valid time 6. Doing decremental valid-time computation, the algorithm reads backwards through the index I_v . Recall that tuples with the same valid time are sorted according to $\text{object}\phi$. Suppose that there are not only tuples for object O_i but also for another object O_j at valid time 6 in our example. Then the algorithm would first process all tuples for object O_i at valid time 6 and then all tuples for object O_j . In the following, we will only consider the tuples of the first object O_i .

In I_v , tuples with the same $vtime$ and $object\phi$ are sorted according to $vtype$ where $END < BEGIN$. Since we are moving backwards through the index, we will first read all tuples with $vtype = BEGIN$ (C_1 - and C_3 -corners) and then all tuples with $vtype = END$ (C_2 - and C_4 -corners) in reverse $ttime$ -order ($ttime$ is the last sorting field in I_v). In figures 6 and 7 the tuples are numbered 1 through 7 in order of occurrence, the type of the corner ($C_1 \dots C_4$) is placed in brackets behind the tuple-number. Before the backlog-tuples for object O_i at valid time 6 are processed, the algorithm assumes that the current timeslice TS is valid with respect to O_i and valid time 6. The algorithm has two steps: one which determines if a validity region has been exited, and a second step to determine if a new region is entered. With decremental valid-time computations, two corners are of interest: the C_1 corner and the C_2 corner.

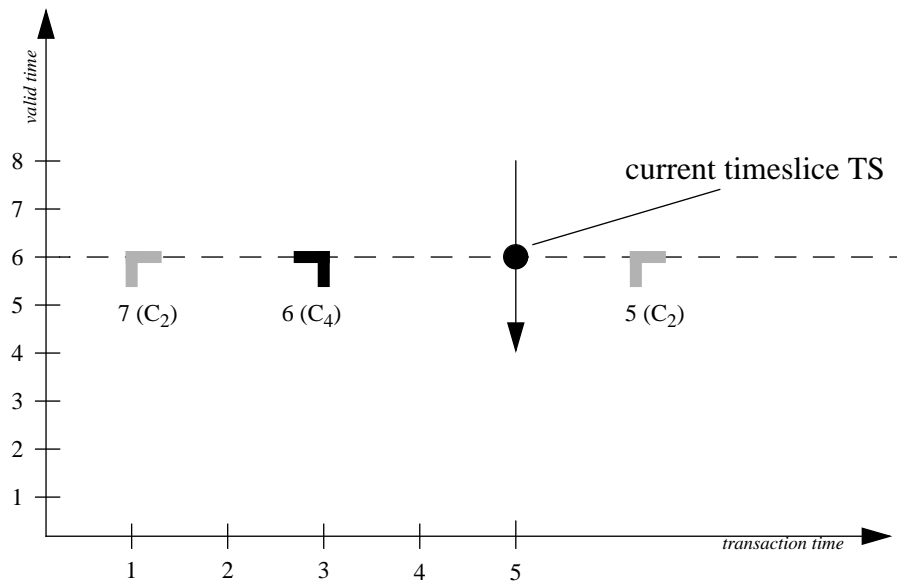
In Figure 6, we have two C_1 and two C_3 corners that have $vtime$ equal to 6. These are processed in the first step of decremental computation that detects whether validity rectangles are exited when crossing valid time 6:

In step 1, tuples with $vtype = BEGIN$ are examined to find out whether a validity rectangle is exited when crossing valid time 6. To do this, the algorithm selects the nearest tuple F that has a transaction time smaller than or equal to the one of the timeslice (tuple 2 in the example).

When decrementing in valid time, a validity rectangle is exited if and only if F is a C_1 -corner ($ttype = INS$).



**Figure 6: Decremental Valid-Time Computation
Step 1: Leaving Validity Rectangles at vtime 6**



**Figure 7: Decremental Valid-Time Computation
Step 2: Entering Validity Rectangles at vtime 6**

In step 2, tuples with $vtype = END$ (C_2 - and C_4 -corners) are examined. Again, the first tuple F with $vtime(F) \leq vtime(TS)$ is selected. We can establish that a validity rectangle is entered during decremental valid-time computations if and only if F is a C_2 -corner (in the example, F is a C_4 -corner). After doing steps 1 and 2 for all objects that have transactions at a certain valid time VT_0 , the timeslice TS is valid for valid time VT_0-1 . If the backlog contains no tuples T with $VT_1 < vtime(T) < VT_0$ then TS is valid for all valid times between VT_1 and VT_0-1 . This is because no validity rectangle can be entered or exited at any of those valid times because there are no corners in that area. Note that TS is also valid at VT_1 because lower bounds belong to a validity rectangle but upper bounds do not. After processing VT_0 , the algorithm can terminate if the desired valid time is between VT_1 and VT_0-1 . Otherwise it continues by processing backlog-tuples with valid time VT_1 until the desired valid time is reached.

Similar arguments can be made for the other three types of computation. In incremental transaction-time computation, the corners of interest are when entering a region, C_1 , and when leaving a region, C_3 . When performing decremental computation through transaction time, one enters a region when one sees a C_3 corner and exits a region when one sees a C_1 corner. Similarly, when one performing incremental computation through valid time, one enters a region whenever one sees C_1 corner, and exits a region at a C_2 corner.

Incremental Valid-Time Computation

The algorithm for incrementing valid time works similar to the decrement algorithm. The differences are:

- I_v is read forward instead of backwards
- The tuple F selected in step 1 and 2 is not the first, but the last tuple with $ttime(F) \leq$

$ttime(TS)$. This is to maintain the property that there are no tuples between F and TS at $vtime(TS)$.

- Validity rectangles are *exited* when C_2 -corners are read and *entered* when C_1 -corners are read.

Proving the correctness of incremental valid-time computation can be done the same way as the previously given proof for the decremental case. This is left as an exercise for the reader.

Differential Transaction-Time Computation

Computation across transaction time can be done the same way as valid-time computation, only the valid-time and transaction-time axis are flipped. Instead of using index $I_V (vtime, object\phi, vtype, ttime)$ we use index $I_T (ttime, object\phi, ttype, vtime)$ where $END < BEGIN$ for $vtype$ and $DEL < INS$ for $ttype$.

The tuple F selected in steps 1 and 2 of incremental/decremental computation is the last/first one read with $vtime(F) \leq vtime(TS)$ instead of $ttime(F) \leq ttime(TS)$ used in valid-time computations. In the incremental case, a validity region is exited if and only if F is a C_3 -corner and a validity region is entered if and only if F is a C_1 -corner in step 2. In the decremental case, it is vice versa: A C_1 -corner in step 1 indicates exiting a validity region and a C_3 -corner in step 2 gives evidence of entering a validity region. Again, the correctness of the algorithm can be proven the same way it was done for valid-time computation.

In Appendix A, we give a procedural representation of all 4 possible movements of a timeslice in the transaction time/valid time grid.

Differential Computation Revisited

Differential timeslice computation was just described as a consecutive crossing of valid or transaction times for which backlog tuples exist. At each crossing, the algorithm first checks whether a validity region is exited for a certain object and then it checks /whether a new validity region is entered for that object. This scheme allowed an easier understanding of the basic concepts of the algorithm but in an actual implementation, performance can be improved by detecting exiting and entering validity regions for all objects together. How this is done is explained below. It will be argued that crossing a valid-time boundary is equivalent to computing two timeslices in a transaction-time relation.

Instead of doing the two steps (detect rectangles that are exited, then detect rectangles that are entered) for one object and then continuing with the next object, it is possible to first do step one for all objects and then do step two for all objects. Step 1 performed for all objects results in a set of (object, item) pairs that have to be deleted from the current timeslice (delete-list) whereas step 2 results in a set of (object, item) pairs that have to be added (add-list).

When doing differential timeslice computation along the transaction-time axis, the add- and delete-lists are fairly small as each transaction gets a distinct timestamp and it only affects one object. The overall performance of the system is unlikely to improve by using a more sophisticated method to compute these lists. For differential valid-time computation it looks different. Any number of transactions can be stored in the backlog concerning the same valid time. For instance, it is likely that a raise in salary is given at the beginning of a year for many employees, each generating a transaction of its own with the same beginning valid time. There might be 10,000 transactions with

valid time “January 1, 1996”, some of them revising outdated salary information. When doing differential valid-time computation across “January 1, 1996”, only those backlog tuples out of the 10,000 that were not invalidated by a later transaction have to be processed. However, when doing differential transaction-time computation across the same 10,000 backlog tuples, there would be 10,000 distinct transaction-time crossings and all the tuples would have to be processed.

The following discussion will only be concerned with crossing valid times. Note, however, that the same techniques can also be used to compute add- and delete-lists when crossing transaction times, but there would not be any savings because a transaction only concerns one object. The backlog tuples needed to compute the add- and delete-lists when crossing some valid time can be described by the following statement:

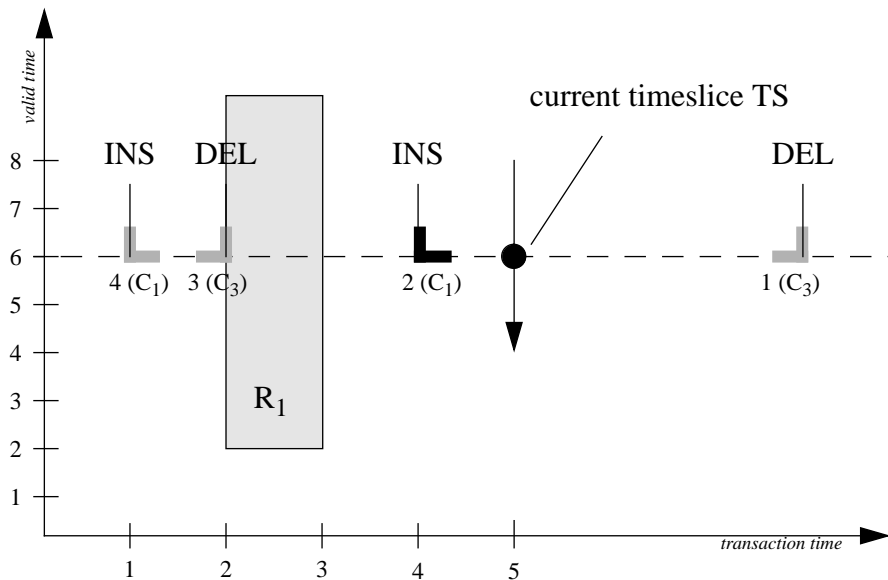
$$BT_{R, VTIME, VTYPE} = \Pi_{item\zeta, object\zeta, ttime, ttype}(\sigma_{vtime = VTIME \wedge vtype = VTYPE}(B_R))$$

For example, $BT_{R, VTIME, VTBEGIN}$ contains the backlog-tuples needed to compute the delete-list when crossing valid time $VTIME$ in decremental computation for the bitemporal relation R . $BT_{R, VTIME, VTEND}$ contains all information necessary for the add-list for the same crossing.

Notice that every $BT_{R, VTIME, VTYPE}$ can be seen as a backlog of a transaction-time relation. It contains add and delete tuples for object-item pairs entered in transaction-time order. It is therefore possible to define a transaction-time timeslice on $BT_{R, VTIME, VTYPE}$: $TS_{R, VTIME, VTYPE}(TTIME)$. What does this timeslice contain? It contains all object-item pairs that have an insert-tuple but no delete-tuple t in

$BT_{R,VTIME,VTEND}$ with $ttime(t) \leq TTIME$. Those are exactly the object-item pairs that have to go into the add/delete-list for incremental valid-time computation.

Figure 8 visualizes this. It is the same example of decremental valid-time computation crossing valid time 6 as used before. The corners in the graph represent tuples in $BT_{R,6,VTBEGIN}$ for one object. The timeslice-operator $TS_{R,6,VTBEGIN}(5)$, of course, takes into account tuples for all objects existing in $BT_{R,6,VTBEGIN}$.



**Figure 8: Decremental Valid-Time Computation
Step 1: Leaving Validity Rectangles at vtime 6**

At first sight it might seem that $TS_{R,VTIME,VTTYPE}(TTIME)$ is the same as $TS_R(TTIME, VTIME)$. Both timeslices lie at the same spot in the transaction time/valid time grid. However, they do not necessarily contain the same tuples. For example, consider $TS_R(2, 6)$ in Figure 8. The chronon (2, 6) is within the boundaries of validity rectangle R_1 , therefore the object-item pair corresponding to R_1 would be in $TS_R(2, 6)$.

However, none of the corners of R_1 has valid time 6, therefore $TS_{R,6,VTBEGIN}$ would not contain this object-item pair.

By first doing step 1 for all objects and then step 2 and by viewing these two steps as transaction-timeslice problems, it is possible to store the views $BT_{R,VTIME,VTTYPE}$ in separate structures optimized for the algorithm used to compute timeslices in transaction-time relations. It is not the topic of this thesis to discuss such algorithms, but it is an important result that such an algorithm can be used as a subroutine for the differential computation of bitemporal timeslices. CHAPTER VI discusses in more detail some implications of this to the storage structure of the valid-time index, CHAPTER VIII discusses how it improves the overall performance and CHAPTER X uses this result to extend the algorithm to n-dimensional space.

CHAPTER VI

STORAGE STRUCTURES FOR BACKLOG-INDICES AND TIMESLICES

This chapter argues that a physical index structure is not necessary for transaction-time index I_T and suggests a nested storage structure for the valid-time index.

Transaction-Time Index

As stated in CHAPTER IV it is possible to store the backlog tuples physically ordered according to index I_T . Therefore, no physical index structure is needed when moving through the backlog in transaction-time order. The only situation where the presented procedures need random access to a backlog tuple using index I_T is in the beginning of differential transaction-time computation:

```
select first tuple, t, s.th. ttime(t) > tt0
```

in incremental and

```
select last tuple, t, s.th. ttime(t) ≤ tt0
```

in decremental transaction-time computation (see CHAPTER V). Notice that the tuple selected in the incremental case is the direct successor of the one selected in the decremental case. The search key in both cases is tt_0 which is derived from the existing timeslice $TS_R(tt_0, vt_0)$ used as the outset for differential computation.

Let ct_i be the current backlog tuple in transaction-time order associated with $TS_R(tt_i, vt_j)$ where ct_i is the last record with $ttime(ct_i) \leq tt_j$. An easy way to find the page where ct_i is located is to store the address of that page with every existing timeslice $TS_R(tt_i, vt_j)$. This enables constant-time access to the tuple needed in the beginning of

differential transaction-time computation. Associating a backlog pointer with each existing timeslice was previously suggested in [22].

When timeslice $TS_R(tt_i, vt_j)$ is incremented/decremented to a new $TS_R(tt_k, vt_j)$ then the address of the corresponding ct_i can be updated to the address of ct_k as well. In the incremental case, ct_k is the last tuple processed and in the decremental case it is the predecessor of the last tuple processed (see CHAPTER V). Therefore, the address of ct_k is directly available at the end of differential transaction-time computation. Notice that ct_i does not have to be updated when doing differential valid-time computation.

The only remaining problem is to get the address of ct_i for timeslices $TS_R(tt_i, 0)$. As stated in CHAPTER V, such timeslices are empty and can be used as the outset for any differential computation. However, the only case where it makes sense to use timeslices $TS_R(tt_i, 0)$ as an outset is incremental valid-time computation. Decrementing valid time is impossible as it is already 0, differential transaction-time computation would leave the timeslice empty. But for incremental valid-time computation, the address of ct_i is not needed. Therefore, the address of ct_i can be set to NULL, indicating that the address is unknown. Should the resulting timeslice be used as an outset later on, the address of ct_i would have to be computed by other means. Assuming that backlog tuples are stored in logically adjacent disk blocks, a possibility would be to perform a binary search on the backlog. Instead of constant time this would yield log-time access. However, this case is very unlikely to happen after the database has been in use for a while. Then, it is much more likely to find a nearby existing timeslice as an outset than using an empty timeslice located along the transaction-time or valid-time axis.

In conclusion, it is possible to efficiently access the backlog in transaction-time order without the use of any index structure. The few cases where a binary search on

the backlog is necessary are negligible as they are only likely to happen shortly after the creation of a bitemporal relation.

Valid-Time Index

As opposed to transaction-time ordering that can be achieved by physically storing the backlog tuples in that order, access to the tuples in valid time order can only be enabled by a dynamic structure that allows out-of-order inserts.

Recall from CHAPTER IV that differential valid-time computation can be solved by computing timeslices in transaction-time backlogs $BT_{R,VTIME,VTYPE}$. Finding the requested transaction-time backlog and indexing the backlog itself can be viewed as separate issues. It is therefore reasonable to use a nested structure similar to the nested ST-tree described in [7].

The first level is a B^+ -tree that only stores *vtime* and *vtype* as key values to provide access to a particular $BT_{R,VTIME,VTYPE}$. The leaves of this tree point to the second level structure that provides access to all tuples of $BT_{R,VTIME,VTYPE}$.

For the second level structure there are two solutions to consider. Which one is optimal depends on the number of tuples in $BT_{R,VTIME,VTYPE}$. However, there is one characteristic common to both approaches: There are no data pointers ultimately pointing to the backlog tuples. Instead, the backlog tuples will be contained in the structure itself, thus duplicating the backlog. This approach is reasonable because the size of a backlog tuple is so small. The simplest solution for the second level structure is to maintain a linked list of disk blocks that give access to the tuples in transaction-time order. As tuples get inserted in this order, this structure is cheap to maintain. The only disadvantage is that all blocks of $BT_{R,VTIME,VTYPE}$ have to be read in order to compute a

timeslice. It can be expected that the number of blocks in these backlogs will usually be very small so that this does not matter. The second solution is to use a structure that allows quick retrieval of a timeslice. This approach is described in the next subsection. The advantage is a better worst-case performance for large backlogs, but there is the disadvantage of the overhead both in storage and computation time introduced by the structure.

Figure 9 visualizes the nested access structure for valid time:

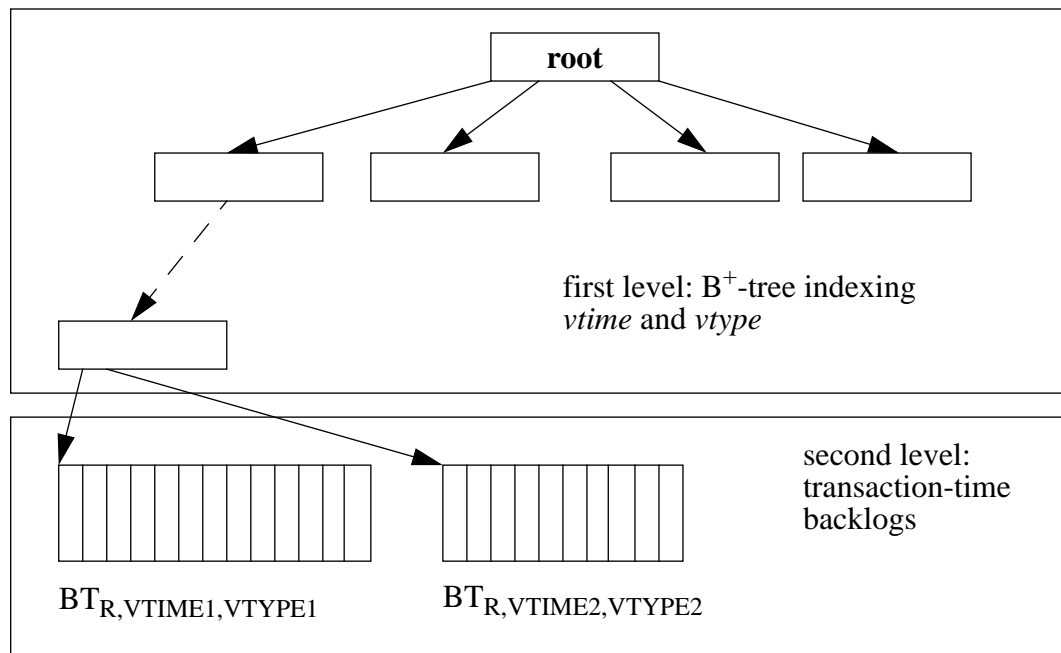


Figure 9: Nested Access Structure for Valid Time

An Alternative Storage Model for $BT_{R, VTIME, VTYPE}$

Besides the naive approach described previously that simply reads the whole backlog $BT_{R, VTIME, VTYPE}$, the transaction-time timeslice problem on these backlogs

can be solved using any existing method. Depending on the method, there will be different storage structures for the backlog.

The incremental implementation model proposed in [11] would be applicable. But in the case of crossing valid times, the efficiency of that model is questionable. Storing previously computed timeslices makes sense only if it is likely that there will be many requests for the same or similar timeslices. Requesting the same timeslice twice would imply that the same valid time is crossed twice at the same transaction time. In the example outlined in CHAPTER V where 10,000 backlog tuples exist for valid time “January 1, 1996”, there might be a cached timeslice $TS_R(tt, \text{“Jan. 10, 1996”})$ which is duplicated and then decremented to $TS_R(tt, \text{“Dec. 15, 1995”})$, crossing valid time “Jan. 1, 1996”. If there is another request for a valid time in December, the second timeslice can be used as an outset, and if January is requested, the first one is used. Only after one of the two timeslices is deleted, another crossing of valid time “Jan. 1, 1996” would be necessary and the previously computed transaction timeslice for that crossing could be useful. But this case is not very frequent. The second scenario where “Jan. 1, 1996” is crossed again is a situation where a timeslice with a valid time in December and transaction time tt' is requested where tt' is far from tt . Then the algorithm might decide to use $TS_R(tt', \text{“Feb. 1, 1996”})$ as an outset instead of $TS_R(tt, \text{“Dec. 15, 1995”})$ and decrement valid time. In this scenario, $TS_R, \text{“Jan. 1, 1996”}, VTYPE(tt)$ would exist from the first crossing and it could be used as an outset to get $TS_R, \text{“Jan. 1, 1996”}, VTYPE(tt')$, but this would not be very efficient because we assumed that tt and tt' are far apart.

Therefore, I propose using a method that accesses the backlog without relying on previously computed timeslices: [23]. The model proposed in [23] allows for constant-time updates and computes a timeslice $TS_{R, VTIME, VTYPE}(TTIME)$ in

$O(|TS_{R,VTIME,VTTYPE}(TTIME)| + \log\log T)$ where $|TS_{R,VTIME,VTTYPE}(TTIME)|$ is the size of the requested timeslice and T is the maximal transaction time. The algorithm maintains an array A where a new entry is appended every time a tuple gets inserted or logically deleted. Every entry in A has the transaction time of its operation associated with it. To index the relation, the entry in A with the highest transaction time before the requested time is used as the starting point. Finding this entry is equivalent to the successor-predecessor problem which can be solved in $O(\log\log T)$ time [23]. Once this starting point is found, the current “access forest” can be traversed, collecting all tuples that are current at the requested time. During the traversal of the access forest, at most twice the number of tuples in the result set have to be accessed.

Storage Models for Timeslices

The bitemporal timeslices proposed in this thesis are a logical extension of the timeslices introduced in [11] for transaction-time databases. For the storage of these timeslices, there are the same options: they can be stored as an index cache or materialized data. An index cache timeslice can be implemented as a relation with the two attributes $object\epsilon$ and $item\epsilon$. $Object\epsilon$ is the primary key as there can be at most one item associated with an object at one chronon in the transaction-time/valid-time grid. [11] makes the assumption that an index cache can generally be kept in main memory. CHAPTER IX of this thesis argues that differential computation in bitemporal databases outperforms other access methods especially for large numbers of objects resulting in large timeslices – and index caches. I will therefore assume in the following that timeslices are generally disk-based structures even in the index cache case.

A materialized timeslice is the result of a join operation between an index-cache timeslice and the item relation (item ϕ is the common key). The result of this join can be stored instead of the index cache.

CHAPTER VII

TIMESLICES FOR TRANSACTION AND VALID-TIME PERIODS

So far the focus of this thesis has been on bitemporal timeslices located at a single chronon in the transaction time/valid time grid. With these timeslices one can run queries like “What was stored in the database on November 20 about the salaries next January?” On the other hand, we would also like to run queries like: “What is currently stored in the database about salaries between January and December 1995?” or “How did the information in the database about salaries in January change over the last 3 Months?” The former query would have to retrieve the valid-time history for 1995 at transaction time NOW while the latter would have to retrieve the transaction-time history for the interval [NOW-3 Months, NOW] at valid time January.

In this chapter, I will extend bitemporal timeslices to capture either valid-time periods for a given transaction time or transaction-time periods for a given valid time. Such timeslices will be called history timeslices. The use of the term “history” is consistent with [10] as the timeslices will either cover the valid-time or transaction-time history of an object. Extensions to the algorithms for differential computations will be given.

Extending Timeslice Relations

The bitemporal timeslices discussed so far were valid at a single chronon in the transaction time/valid time grid. Therefore, there was no need to store any timestamp-attributes in the timeslice-relations. Each timeslice was associated with a single trans-

action and valid time. Now the timeslice-relations have to store the object-history on one time-dimension, either transaction or valid time. The second dimension of time will again be a fixed value. There is no need to have separate data-models for transaction-time and valid-time histories. The type of information stored in both cases is the same. The only thing needed is a flag associated with a timeslice that indicates whether it is a transaction or valid-time history. The interval of the historically stored time dimension and the fixed value of the second dimension will also be associated with the timeslice. The following discussion will make implicit use of this additional information by referring to timeslices as $TS_R(ttbegin\ to\ ttend,\ vtime)$ for transaction-time histories and $TS_R(ttime,\ vtbegin\ to\ vtend)$ for valid-time histories. The begin and end attributes in the relation correspond to the historically kept time dimension. Table 4 lists all attributes of such timeslices.

Table 4: Attributes of History Timeslices $TS_R(ttbegin\ to\ ttend,\ vtime)$ and $TS_R(ttime,\ vtbegin\ to\ vtend)$

attribute	domain
object ζ	surrogate
item ζ	surrogate
begin	timestamp
end	timestamp

As usual, begin-times are included whereas end-times are not included in validity periods. There will also be a special value UB for begin and end times indicating that the validity period is unbounded within the bounds of the timeslice. For example, begin would be set to UB if the validity of a timeslice-tuple would start before vtbegin in a timeslice $TS_R(ttime,\ vtbegin\ to\ vtend)$. Some tuples might start being valid before vt-

begin and continue being valid after $vtend$. In this case, both begin and end would be set to UB.

Primary key for history timeslices is $(object\phi, begin)$. Validity regions concerning one object are not allowed to overlap so $object\phi$ together with begin is a key candidate. A second key candidate is $(object\phi, end)$ for the same reason.

Again, a history timeslice can either be stored as an index-cache with solely the attributes just described or it can be stored as materialized data including all user-defined attributes from the item relation T_R .

Defining Differential Computation for History Timeslices

Before describing extensions to differential computation for history timeslices, it must be clear how such timeslices can be moved in the transaction time/valid time grid. It should be quite obvious how history timeslices can be moved in the direction of the time-axis that has a fixed value. For example, we might want to increment timeslice $TS_R(ttime, vtbegin \text{ to } vtend)$ to $TS_R(ttime + offset, vtbegin \text{ to } vtend)$. But what does it mean to move that timeslice along the valid-time axis? For simplicity, it will only be allowed to change either $vtbegin$ or $vtend$, thus extending or shrinking the valid-time period. $TS_R(ttime, vtbegin \text{ to } vtend)$ can be incremented to $TS_R(ttime, vtbegin \text{ to } vtend + offset)$. This still allows us to move the whole valid-time period without changing the length of it, it would only have to take place in two steps: first move $vtend$ and then $vtbegin$ in case of incremental computation.

Some work has been done to evaluate time-varying query expressions for valid-time relations [2]. For example, such a query might ask for all tuples with valid times between “NOW – 3 Months” and “NOW”. The result is similar to a history timeslice.

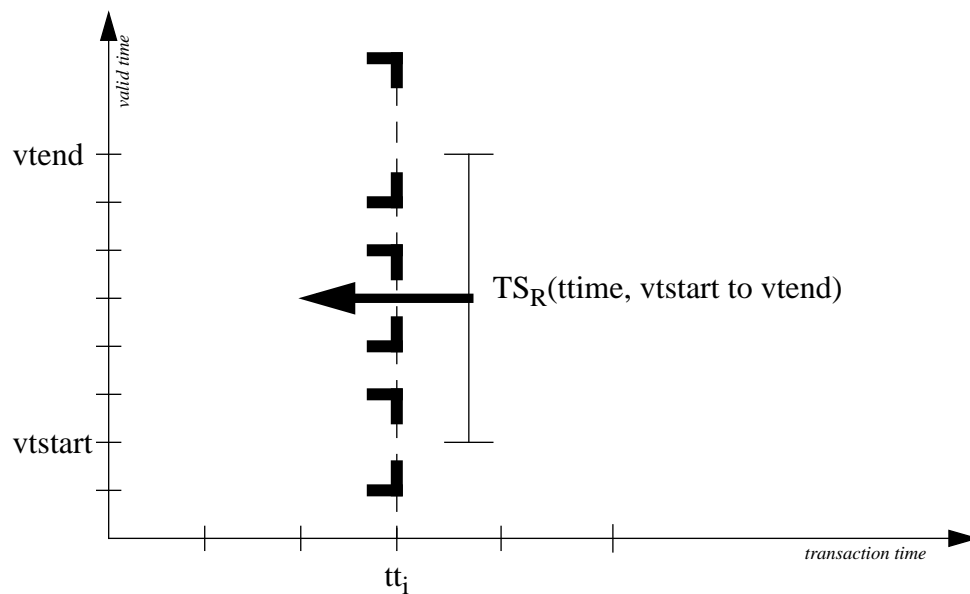
In [2] a technique for how the cached query result can be updated as the value of NOW changes is described. This corresponds to incremental computation on history timeslices. However, the algorithm presented in [2] is applicable to incremental computation only. The differential algorithms presented here are more general as they work with any type of differential computation in bitemporal space.

Extending Differential Computation

I will first show how an extended algorithm does differential timeslice computation along the axis for which the history timeslice has a fixed value. Figure 10 is an example where timeslice $TS_R(\text{time}, \text{vbegin to vtend})$ is decremented in transaction time. Again, the algorithm consecutively crosses transaction-time boundaries and for each crossing it is first detected whether validity rectangles are exited and then whether new rectangles are entered. Figure 10 shows the second step of finding rectangles that are entered.

For standard bitemporal timeslices it was sufficient to test for each object whether there is a validity rectangle starting at the crossing's transaction time tt_i that covers the valid time of the timeslice. For history timeslices, one has to find all validity rectangles starting at tt_i that intersect with the valid-time period of the timeslice $[\text{vstart}, \text{vtend}]$. As the algorithm has access to backlog-tuples in valid-time order, it is relatively simple to find these rectangles. Going backwards (or forward: in incremental computation) in valid time, the algorithm can collect all pairs of backlog-tuples (t_1, t_2) that span the same validity rectangle where $\text{vtype}(t_1) = \text{vbegin}$, $\text{vtype}(t_2) = \text{vtend}$ and the valid-time period covered by t_1 and t_2 intersects with the timeslice: $\text{vtime}(t_1) < \text{vtend}$ and $\text{vtime}(t_2) > \text{vbegin}$. Note that a tuple t_2 does not exist if the rectangle is unbounded in

valid time. However, this is easy to detect and conceptually one can think of t_2 as being a tuple with valid time “Forever”. For every such pair (t_1, t_2) the algorithm inserts a new tuple into the timeslice. Object ϕ and item ϕ are set according to the values in t_1 and t_2 , begin is set to $vtime(t_1)$ if $vtime(t_1) \geq vtbegin$, otherwise it is set to UB. Similarly, end is set to $vtime(t_2)$ if $vtime(t_2) \leq vtend$ and UB otherwise. Exiting validity rectangles in step 1 can be detected the same way. The algorithm again collects pairs (t_1, t_2) of back-log-tuples and uses them to identify timeslice-tuples that have to be deleted.



**Figure 10: Decrementing Transaction Time for $TS_R(ttime, vtstart \text{ to } vtend)$
Step 2: Entering Validity Rectangles**

Extending History Timeslices

Extending a history timeslice means either increasing its end-time or decreasing its begin-time. This is done by taking the desired end-point and moving it to its new position in the transaction time/valid time grid. Every time a transaction time/valid time

boundary is crossed, the timeslice is updated like in all other cases of differential computation. Figure 11 shows how the end-time of a transaction-time history is incremented.

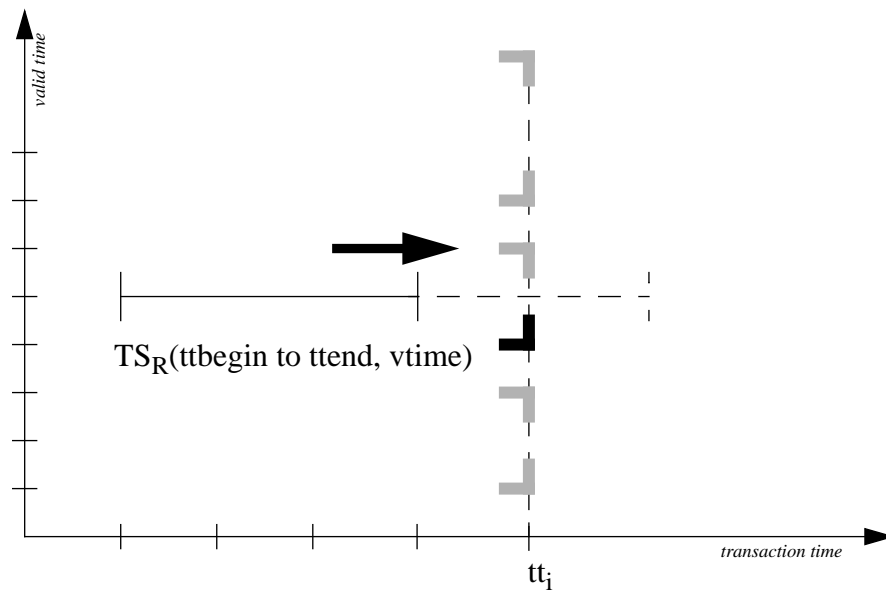


Figure 11: Extending $TS_R(ttbegin\ to\ ttend, vtime)$

When a validity rectangle is exited like in this example then the (object, item) pair of the rectangle is already stored in some tuple t in the timeslice because its lower transaction-time boundary is before the current end-time of the timeslice. However, $end(t)$ is set to UB because the upper bound was beyond the end-time of the timeslice. The only thing the algorithm has to do in this case is to select tuple t in the timeslice and set $end(t)$ to tt_i (the transaction time being crossed). In order to find t , one can traverse the B^+ -tree which is assumed to exist for the primary key (object ϕ , begin) and select the last tuple where object ϕ corresponds to object ϕ of the corner being processed. When

decrementing the begin-time of the timeslice, the value of $\text{begin}(t)$ has to be updated similarly.

When a new validity rectangle is entered then there is no entry for this rectangle in the timeslice because the lower bound of the rectangle is beyond the current end-time of the timeslice. Therefore, a new tuple has to be inserted into the timeslice, $\text{object}\phi$ and $\text{item}\phi$ are set according to the corners, begin is set to the valid/transaction time being crossed and end is set to UB because the upper bound of the rectangle is still beyond the end-time of the timeslice.

Notice that transaction/valid time boundaries are crossed at a single instant in valid/transaction time. It is therefore sufficient to detect those corners that cover this chronon and the optimizations proposed in CHAPTER VI can be used. This is not possible when moving history timeslices along the axis for which they have a fixed value.

Shrinking History Timeslices

When shrinking history timeslices, (object, item)-pairs have to be deleted when a validity rectangle is exited and the begin/end -value has to be set to UB when validity rectangles are entered. It is exactly the reverse process to the extension of history timeslices.

CHAPTER VIII

COMPLEXITY OF THE ALGORITHM

This chapter analyses the complexities of the algorithms introduced in CHAPTER V for updating and querying bitemporal relations. In comparison with tree-based methods, differential computation has a different set of cost-variables. The size of the backlog-part which is traversed plays a much more important role than the number of tuples stored in the backlog.

Update Processing

This discussion of update processing includes insert, delete and update operations. Recall from CHAPTER V that delete and insert are the basic operations and that update is simply a sequence of those two operations. The following variables will be used to describe the complexity of an operation for object O and valid-time period $[vt_B, vt_E]$:

- H : number of tuples in the current valid-time history $TS_R(\text{NOW}, 0 \text{ to Forever})$
- h : number of tuples in $TS_R(\text{NOW}, 0 \text{ to Forever})$ for object O valid during $[vt_B, vt_E]$
- V : number of distinct valid-time instants in B_R .

There are three steps in the delete-operation that interleave each other: Tuples in $TS_R(\text{NOW}, 0 \text{ to Forever})$ for object O intersecting valid-time period $[vt_B, vt_E]$ have to be selected, then backlog-tuples are appended that indicate the logical deletion of those tuples and finally the timeslice is updated as we assume eager maintenance of the current valid-time history. These three steps have the following complexity:

1. Assuming that a B^+ -tree is maintained for the primary key (object ϕ , begin) of the timeslice, it takes $O(\log H)$ time to find the first tuple that intersects with $[vt_B, vt_E]$ and $O(h)$ to read the rest of the desired tuples resulting in a total of **$O(h + \log H)$** .
2. For all h tuples selected in the current valid-time history, between one and four new tuples have to be stored in the backlog and inserted into index I_V (see CHAPTER V). In both suggested methods, the upper-level tree has to be traversed which takes $O(\log V)$ time for each insertion, whereas only constant time is needed for the second level.

The total time for step 2 is therefore **$O(h \log V)$** .

3. All h tuples selected in the current timeslice either have to be modified or removed. The complexity for that is **$O(h \log H)$** .

The total complexity of the delete-operation is therefore:

$$O(h + \log H) + O(h \log V) + O(h \log H) = O(h(\log H + \log V)).$$

In most cases, $H < V$ can be expected leading to **$O(h \log V)$**

The insert-operation takes $O(\log V)$ time. This is the required time to insert two tuples into the backlog. Appending a tuple to the item relation takes constant time as there are no indices defined on that relation.

Summarizing the complexities of all operations we get the following:

- Insert: $O(\log V)$
- Delete, Update: $O(h \log V)$

Query Processing

This thesis is concerned about differential timeslice computation. However, computing a timeslice is only part of answering a query in a temporal database. If the

query is a pure timeslice query requesting *all* objects and their corresponding items at a given chronon in the transaction time/valid time grid, then the whole timeslice must be given as output. In cases where the timeslice is big and the differential computation small, this output will be the major part of the complexity. In the more general case of a range timeslice query, only part of the timeslice is given as output. If there is an additional index-structure on a materialized timeslice, this might be very fast. Computational complexity arising from steps after the timeslice-computation is hard to predict. It is not the topic of this thesis. In the following, only the costs of differentially computing a new timeslice from an outset will be examined. Additional costs that arise when the base timeslice is first duplicated are not counted.

Cost-variables for differential timeslice computation:

- D: number of tuples in the “differential file” [11], i.e. the number of tuples that are in the backlog whose transaction/valid times are in the region browsed during differential transaction/valid time computation.
- S: number of tuples in the timeslice which is incremented/decremented.
- U: number of updates that become necessary in the timeslice ($U \leq D$)
- C: number of transaction/valid time crossings: the number of distinct transaction/valid time instants with corresponding backlog-tuples in the time-period of the computation. ($C \leq D$)
- T: Maximum value for transaction-times

The first step in differential computation is to find the page in the index I_V (for valid-time computation) or the backlog (for transaction-time computation) where the computation begins. As mentioned in CHAPTER VI, the current backlog page can be stored with each timeslice so the page can be found in constant time for transaction-time

computations. For valid times, however, this is not possible because index I_V allows inserts in any order resulting in tuples being moved to a different page when nodes are split (see [3]). It therefore takes $O(\log V)$ time to find the starting page with V being the number of valid-time instants as defined for update processing.

In the standard case it takes $O(D)$ time to find all updates (inserts, deletes) that are necessary in the timeslice. This is because all backlog-tuples in the transaction/valid time period which is examined are read. Notice that this complexity is the same for transaction and valid-time computation. Once the first page in I_V is accessed, succeeding pages can be found in constant time. In the optimized case, it takes $O(\log \log T + u_c)$ time to get all necessary updates for one crossing c using the algorithm proposed in [23], u_c is the number of updates for this one crossing. The total amount of work for all crossings is therefore $O(C \log \log T + U)$ in the optimized case.

Finally, it takes $O(\log S)$ time for each update to be stored in the timeslice. This time bound is the same for both index-cached and materialized timeslices. As opposed to index-cached timeslices, attribute data has to be retrieved from the item relation and multiple indices might have to be updated for materialized timeslices. As item surrogates are implemented as increasing integers, the item relation can be accessed in constant time and updating multiple indices only changes the constant in front of $\log S$ which is not considered in the O -notation. The time for updating the timeslice is therefore $O(u \log S)$.

The total complexity for differential computation is:

$O(D + u \log S)$ standard, $O(C \log \log T + u \log S)$ optimized. $O(\log V)$ has to be added for valid-time computations in both cases.

Clearly, the efficiency of differential computation crucially depends on minimizing the value of D by picking the optimal existing timeslice as a base. For the transaction-time dimension, the techniques presented in [22] can be used to efficiently find the optimal timeslice at the expense of maintaining a physical index-structure for the transaction-time index I_T . This does not deteriorate update-complexity as $O(\log N)$ is contained in updates anyway. It is a topic of future research to explore the possibilities of using a similar technique for the valid-time index.

CHAPTER IX

COMPARISON OF THE PROPOSED ALGORITHM WITH EXISTING SOLUTIONS

Thus far there are two approaches to the computation of bitemporal timeslices published in literature: [17] and [15]. The two approaches are conceptually different as [17] uses differential computation to reconstruct old database states, whereas [15] uses a special tree-structure that allows the retrieval of any database state without differential computation.

The following two sections give a comparison of the explicit invalidation model with the existing models. It will be shown that the M-IVTT tree proposed in [17] can be emulated by the explicit invalidation model by applying a certain strategy for the storage of timeslices. The bitemporal indexing tree introduced in [15] has good worst-case performance but it has disadvantages over the explicit invalidation model in many practical cases. The final argument will be that a combination of a direct approach like [15] and the implicit invalidation model is likely to be successful in many applications.

M-IVTT versus Explicit Invalidation Model

The Multiple Incremental Valid Time Tree (M-IVTT) proposed in [17] stores a certain number of valid-time trees (VTTs) that are current at some instant in transaction time. Timeslices are computed by choosing a nearby VTT (i.e. one whose transaction time is near the requested transaction time) and incrementally or decrementally apply-

ing “patches” that represent the update activity in the relation to get the VTT for the desired transaction time. The requested timeslice is the result of a valid-time selection on this tree.

A VTT for transaction time tt in the M-IVTT model contains the same data as a history timeslice $TS_R(tt, 0 \text{ to Forever})$ in the explicit invalidation model. The “patches” used in M-IVTT are just a different representation of the data stored in the backlog proposed here. The M-IVTT approach can therefore be viewed as a special case of the more general model introduced here.

Bitemporal Interval Tree versus Explicit Invalidation Model

Using the Bitemporal Interval Tree [15], a bitemporal pure timeslice query can be solved in $O(\log V + \log N + A)$ time where V is the number of valid-time instants the index is “focused” on, N is the number of changes recorded for the relation and A is the answer size. In the worst case, this performance is much better than $O(C \log \log T + U \log S)$ when using optimized differential computation because the number of transaction/valid time crossings C can get very big in the worst case.

In an average case scenario, however, C will often be rather small. For example, it is likely that the state of a company’s finances at the end of a fiscal year are queried many times or there might be some investigation of the update activity in the database during a short period of (transaction) time when a new apprentice got access to the database.

In cases where C is small, it is faster to read a small fraction of the backlog and apply changes to an existing timeslice than traversing a bulky structure like the Bitemporal Interval Tree where all update-activity is stored. Moreover, the O -notation hides

the fact that a page-fault can occur for every single result-tuple when querying the Bitemporal Interval Tree.

In the more general range timeslice query or in join-operations with bitemporal relations, the advantages of differential computation can be even bigger. In those cases, a bitemporal timeslice is just an intermediate result and its size can be much bigger than the size of the final result. If there is a complex query involving one or more bitemporal relations then the result of this query can be cached. Suppose there is a new query that has the same join-, union- and select-operators as a previously cached query with the only difference that the valid or transaction times of an involved bitemporal relation slightly differs. Then differential computation can be used to compute add- and delete-lists for the involved bitemporal relations. These lists can be used to incrementally update the query-result using methods introduced for conventional databases (see [5], [18]). This is likely to be much faster than re-computing the whole query which would be necessary if Bitemporal Interval Trees were used.

Despite all advantages of differential computation one has to take into account that other access methods are faster if no outset near the target timeslice exists, i.e. decent worst-case performance can not be guaranteed by using differential computation alone. In order to improve worst-case performance, [17] suggests storing valid-time trees every D_t transactions. For large databases, D_t has to be rather big to limit space consumption, again resulting in poor worst-case performance. Moreover, most of the stored timeslices are unlikely to be used. It is the authors opinion that the best overall performance can be achieved by combining a direct access method like the Bitemporal Interval Tree with the implicit invalidation model introduced here. The direct method would be useful when there is no nearby cached timeslice and differential computation

otherwise. Obviously, the two methods can be combined by maintaining separate data structures for both models. It is a topic of future research to find ways to combine the two methods without duplicating the whole data.

Table 5 summarizes the advantages and disadvantages of the three discussed algorithms:

Table 5: Summary of Existing Bitemporal Timeslice Algorithms

Model	Advantages	Disadvantages
BIT	<ul style="list-style-type: none"> good worst-case performance 	<ul style="list-style-type: none"> focused on a valid-time period, less efficient if a valid time outside the focus is requested poor average-case performance
M-IVTT	<ul style="list-style-type: none"> relatively good worst case relatively good average case 	<ul style="list-style-type: none"> requires very much disk-space for acceptable worst case average case worse than exp. invalidation model (always have to traverse VTT)
exp. invalidation model	<ul style="list-style-type: none"> good average-case performance 	<ul style="list-style-type: none"> poor performance if good out-set does not exist

Comparing bitemporal timeslice algorithms – an example

In order to obtain a better understanding of cases where the presented algorithms show their strengths and weaknesses, we will discuss the run-time of pure-timeslice queries on a sample bitemporal relation.

Consider the relation `emp_proj(employee-ID, project-ID)` that assigns employees to projects during some period in valid time. Projects may last any length of time with an average of 1 month. Tentative project assignments are made well in advance resulting in frequent changes. On average, there is one update per project assignment. Suppose that the relation `emp_proj` was created 5 years ago and it contains project as-

signments for a 5-year period. There are $5 \times 12 = 60$ project assignments per employee for the 5-year valid-time period. Since there is one update per assignment, the number of transactions stored in this relation is 120 per employee. With 100,000 employees, there is a total of 12 million transactions in the relation emp_proj. Table 6 summarizes the characteristics of the sample relation.

Table 6: Sample Relation emp_proj

project assignments per employee	60
transactions per employee	120
number of employees	100,000
total transactions in backlog/BIT	12,000,000
average number of entries in VTT	6,000,000
entries in bitemporal timeslice	100,000

The least space to store the 12 million transactions is required in the M-IVTT approach. The transactions are directly appended to the backlog and the only access structure is the transaction-time tree. The implicit invalidation model requires about twice as much space since the backlog is replicated in the valid-time index. Experiments presented in [15] show that the BIT requires roughly 4 times as much space as the R-tree. Therefore, the space requirement for the BIT is at least 4 times as high as for the M-IVTT.

Run-time of queries

Assuming that the valid time for all queries is within the focus of the BIT, the time to process a query is not influenced by the valid/transaction time of the query. The picture looks different in the M-IVTT case. If there is a VTT for the requested transaction time, only that tree has to be queried. As there is one update per project assignment,

the number of entries in the VTT is about half the amount of entries in the BIT (see Table 6). It is therefore reasonable to assume that M-IVTT is clearly faster than BIT in this case. The question is: How many updates can be applied to an existing VTT without exceeding the runtime of a BIT query? Both queries and updates to a VTT require logarithmic time, the same is true for accesses to the BIT. Since the BIT contains only about twice as many entries as a VTT, the number of updates that can be done in the VTT in the given time frame is quite small. Even with extensive caching it is unreasonable to believe that more than 100 updates can be applied to the VTT without exceeding the BIT runtime in our example. We can easily compute how much time passes before 100 transactions are entered into `emp_proj`: Our business is open 8 hours a day, 5 days a week and 50 weeks a year. In 5 years that is a total of 10,000 hours of operation. That makes an average of 1,200 transactions per hour. 100 transactions cover only 5 minutes in the life of the relation `emp_proj`! In other words: The M-IVTT approach is only faster than BIT if there is an existing VTT whose transaction time is within 5 minutes of the requested time. The discussion shows that it is probably not reasonable to use M-IVTT for this relation. The reason for this poor relative performance is evidently the big size of the VTT compared to the total amount of data. M-IVTT is only useful when there is much more update activity on existing data.

In the explicit invalidation model, it might also be necessary to update an existing timeslice to get the requested one. However, a bitemporal timeslice has only one entry per employee, making a total of 100,000 entries. Since a timeslice entry contains only the object-ID and item-ID, it is reasonable to assume that the timeslice can be entirely kept in main memory. Updates to the timeslice do not require disk access. It is therefore sufficient to count the number of transactions that can be read from the back-

log before the run-time of a BIT query is exceeded. For differential computation in transaction-time order, this number is quite big because disk pages in both the transaction-time tree and the backlog are completely full and the tuples are physically stored in transaction-time order. Since backlog tuples are small there can be thousands of transactions stored in one page. It is therefore reasonable to estimate that 100,000 transactions can be processed before the run-time of the BIT is reached. For valid-time computations, I estimate 50,000 transactions because leaf nodes in the valid-time tree are at least half full. Comparing these numbers to the update activity on the relation, we get a maximum distance to an existing timeslice of about one week in valid time or two weeks in transaction time. If the distance is bigger, the runtime of the BIT is exceeded.

These numbers imply that it is indeed useful to employ the explicit invalidation model in this example. It is likely that there will be many queries requesting valid and transaction times near NOW. In that case, a relatively small number of cached timeslices is enough to provide outsets with a distance of only a few days to the transaction/valid time of most queries. It is also possible to store a large number of timeslices because they are very small compared to the size of the backlog. In a case where the nearest timeslice is months away from the queried time, however, the runtime of the query becomes a multiple of the BIT runtime. Table 7 summarizes the results.

Table 7: Runtimes of a Pure-Timeslice Query on the Sample Relation

model	runtime
BIT	independent of queried valid/transaction time: C steps
M-IVTT	runtime $< C$ if an existing VTT is at most 5 minutes apart from the queried transaction tim
exp. invalidation model	runtime $< C$ if an existing timeslice is at most 2 weeks apart from the queried transaction time or 1 week apart from the queried valid time (or combinations thereof)

CHAPTER X

EXTENDING THE ALGORITHM TO N-DIMENSIONAL SPACE

CHAPTER V described a method how crossing transaction or valid times can be solved by computing two one-dimensional timeslices. This chapter will give an example of an application where more than 2 dimensions of time are of interest. It will be shown that differential computation can be applied to n-dimensional space by moving along one dimension at a time and consecutively crossing boundaries like in bitemporal space. These crossings can be reduced to (n-1)-dimensional timeslice problems.

Justification of Higher Dimensions of Time – an Example

The example described here has been previously presented in [6] to show that transaction time and valid time are not always sufficient to model all aspects of time needed in an application. In order to process income support claims, the UK department of social security (DSS) has to know certain circumstances of applicants such as the number of dependants, savings and financial commitments. These circumstances are true during some period of time (valid time). It also must be stored when the DSS first believes a circumstance to be true and when it ceases to believe in it. I will refer to these times as “reporting time”. It is noted in [6] that “reporting time” is not necessarily the same as the time when the data is actually stored in the database. For example, the reporting time could be defined as the day when mail concerning circumstances arrives. On the other hand, the data might not be entered into the database until the information is verified. Therefore, transaction time is needed as a third dimension of time.

3-dimensional Differential Computation

In the above example, a timeslice has three dimensions: $TS_R(ttime, vtime, reptime)$. It returns all circumstances that were current in the database at *ttime*, valid at *vtime* and believed to be true by the DSS at *reptime* (report time). Differential computation is again applicable to one dimension at a time. The given timeslice could, for example, be incremented in *reptime* to $TS_R(ttime, vtime, reptime + offset)$.

The temporal elements associated with the stored facts have three dimensions. Instead of representing them by validity rectangles, validity cuboids are used. This can be done using a straightforward extension of the backlog introduced in CHAPTER IV. For the additional time dimension, attributes “*reptime*” and “*rtype*” are needed. These are analogous to “*vtime*”, “*vtype*” and “*ttime*”, “*ttype*”, respectively. Backlog tuples now represent corners of validity cuboids. There are eight types of corners identified by the combinations of the begin/end flags of the three time dimensions: $\{ INS, DEL \} \times \{ BEGIN, END \} \times \{ REP_BEGIN, REP_END \}$.

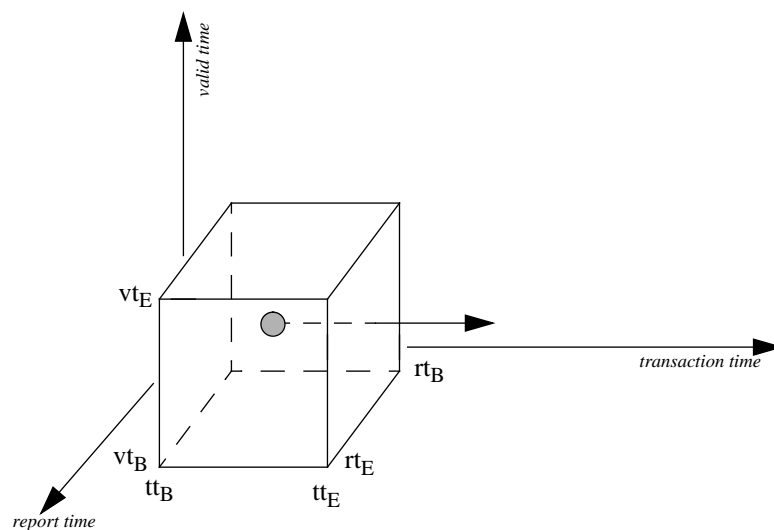


Figure 12: A Validity Cuboid

Figure 12 shows the validity cuboid of a fact which is valid between vt_B and vt_E , believed to be true between rt_B and rt_E and current in the database between tt_B and tt_E . The dot inside the cuboid represents an existing timeslice. This timeslice is incremented in transaction time. Like in the bitemporal case, the algorithm looks for transaction-time crossings, i.e. transaction times for which backlog tuples exist. In Figure 12, this is true at transaction time tt_E . Again, there are two phases: in phase 1, all tuples with $ttype = DEL$ are examined to find cuboids that are exited and in phase 2, validity cuboids that are entered are found by looking at tuples with $ttype = INS$. In both phases, the examined tuples lie in the plane described by the equation *transaction time* = tt_E . Reduced by transaction time, backlog tuples can be seen as corners of validity rectangles. The task is to find all validity rectangles that span (rt_{TS}, vt_{TS}) , the report and transaction time of the timeslice. We have thus reduced three-dimensional differential computation to solving the two-dimensional timeslice problem twice for every crossing.

Generalization to n Dimensions

The differential computation scheme described for two and three dimensions can be extended to n dimensions as follows:

For each dimension $i = 1 \dots n$ the backlog gets attributes $time_i$ and $type_i$. The attributes $time_i$ store the location of the tuple in n-dimensional space, $type_i$ is a flag that indicates whether $time_i$ is a beginning or an ending time, possible values are $BEGIN_i$ and END_i .

Differential computation in dimension j to transform $TS_R(t_1, \dots, t_j, \dots, t_n)$ to $TS_R(t_1, \dots, t_j + \text{offset}, \dots, t_n)$ is accomplished by consecutively processing crossings in

time dimension j . Like in the 2- and 3-dimensional case, a crossing of time $t_{j,c}$ in the j -dimension can be reduced to two $(n-1)$ -dimensional timeslice computations. The set of backlog-tuples needed for this computation is:

$$\prod_{\text{time}_1, \text{type}_1, \dots, \text{time}_{j-1}, \text{type}_{j-1}, \text{time}_{j+1}, \text{type}_{j+1}, \dots, \text{time}_n, \text{type}_n} (\sigma_{\text{time}_j = t_{j,c} \wedge \text{type}_j = \text{TYPE}} (\mathbf{B}_R))$$

In incremental computation, TYPE must be END_j to get the delete-list and BEGIN_j to get the add-list for the crossing. In decremental computation it is vice versa.

The algorithms outlined here require the backlog to be consistent with all facts entered into the database. Like in the bitemporal case, this requires a fair amount of work when entering new facts. All facts that are current and overlap with the new fact have to be invalidated. An efficient implementation of this for n dimensions is a subject of future research. Also, moving along time axes during differential computation requires an index structure for every dimension. This is another topic of future research.

CHAPTER XI

CONCLUSIONS

In this thesis, I have presented a method for applying differential timeslice computation to bitemporal relations. The method is a logical extension of the differential computation for transaction-time relations introduced by Jensen. The two main differences are that an additional index structure is needed for valid time and that crossing a point in valid time or transaction time requires the computation of two single-dimensional timeslices. This generalization allows the application of the algorithm even for higher dimensions of time. For every additional time axis, an additional index structure is needed. Crossing a point in n -dimensional time requires the computation of two $(n-1)$ dimensional timeslice. History timeslices were introduced as another generalization extending the validity of bitemporal timeslices to time periods.

It was pointed out that the computation of the single-dimensional timeslices needed for the crossings in bitemporal space is a separate issue. The naive approach used in the first description of the algorithm simply reads all backlog tuples of a particular crossing. As these timeslices are rather unlikely to be reused, it was suggested to alternatively use the direct approach of Faloutsos to compute them. This gives a better asymptotic performance of the algorithm. However, it remains an interesting open question how large the data has to be for Faloutsos' algorithm to have advantages over the naive approach. For small amounts of data, as expected for transaction-time crossings, the naive approach is better because it has much less overhead.

The analysis of the algorithm's complexity has shown that updates are possible in logarithmic time. The time required to differentially compute timeslices is essentially linear in the number of crossings plus the time necessary to update the timeslice. The overall performance highly depends on the average number of crossings per timeslice computation. This, in turn, depends on the existence of suitable outsets and query-patterns. A direct comparison to direct bitemporal access methods is not possible, but there are clear scenarios where differential computation and direct methods perform better, respectively. Therefore, a combination of both methods is most likely to give optimal performance. A space-efficient combination of two such methods is a topic of future research. Another open question is how the best outset is chosen among a set of existing bitemporal timeslices. Storage strategies for timeslices are an interesting, orthogonal issue: How many timeslices are cached? Which one is deleted when the maximum number is reached?

APPENDIX A

PSEUDO CODE FOR DIFFERENTIAL BITEMPORAL TIMESLICE COMPUTATION

Incremental Transaction-Time Computation

```

procedure increment-time( $TS_R(tt_0, vt_0, tt_D)$ )
  use physical ordering of the backlog (index  $I_T$ )
  select first record  $t$  s.th.  $ttime(t) > tt_0$ 
  while  $ttime(t) \leq tt_D$  do
    while  $vtime(t) > vt_0$  do ;; ignore irrelevant records
       $t := next-record(t)$ 
    od

    if  $ttype(t) = DEL$  then ;; detect whether a validity rectangle is exited
       $tt := t$ 
      ;; find the delete-record pointing to the same object with
      ;; valid time closest to  $vt_0$ 
      while  $object\phi(t) = object\phi(tt)$  and  $ttime(t) = ttime(tt)$ 
        and  $ttype(tt) = DEL$  and  $vtime(tt) \leq vt_0$  do
           $t := tt$ 
           $tt := next-record(tt)$ 
        od
      ;; if the record with closest valid time is a  $C_3$ -corner, delete the corresponding item,
      ;; otherwise ignore
      if  $vtype(t) = BEGIN$  then
        delete  $object\phi(t)$  from timeslice
      fi
    else ;;  $ttype(t) = INS$ , detect whether a validity record is being entered
      ;; find the insert-record pointing to the same object with
      ;; valid time closest to  $vt_0$ 
       $tt := t$ 
      while  $object\phi(t) = object\phi(tt)$  and  $ttime(t) = ttime(tt)$ 
        and  $ttype(tt) = INS$  and  $vtime(tt) \leq vt_0$  do
           $t := tt$ 
           $tt := next-record(tt)$ 
        od
      ;; if the record with closest valid time is a  $C_1$ -corner, insert the corresponding item,
      ;; otherwise ignore
      if  $vtype(t) = BEGIN$  then
        insert into timeslice ( $object\phi(t)$ ,  $item\phi(t)$ )
      fi
    fi
     $t := next-record(t)$ 
  od
end

```

Incremental Valid-Time Computation

```

procedure increment-vtime( $TS_R(tt_0, vt_0), vt_D$ )
  use index  $I_V$ 
  select first record  $t$  s.th.  $vtime(t) > vt_0$ 
  while  $vtime(t) \leq vt_D$  do
    while  $ttime(t) > tt_0$  do ;; ignore irrelevant records
       $t := next\_record(t)$ 
    od

    if  $vtype(t) = END$  then ;; detect whether a validity rectangle is exited
       $tt := t$ 
      ;; find the END-record pointing to the same object with
      ;; transaction time closest to  $tt_0$ 
      while  $object\phi(t) = object\phi(tt)$  and  $vtime(t) = vtime(tt)$ 
        and  $vtype(tt) = END$  and  $ttime(tt) \leq tt_0$  do
           $t := tt$ 
           $tt := next\_record(tt)$ 
        od
      ;; if the record with closest transaction time is a  $C_2$ -corner, delete the corresponding item,
      ;; otherwise ignore
      if  $ttype(t) = INS$  then
        delete  $object\phi(t)$  from timeslice
      fi
    else ;;  $vtype(t) = BEGIN$ , detect whether a validity record is being entered
      ;; find the insert-record pointing to the same object with
      ;; transaction time closest to  $tt_0$ 
       $tt := t$ 
      while  $object\phi(t) = object\phi(tt)$  and  $vtime(t) = vtime(tt)$ 
        and  $vtype(tt) = BEGIN$  and  $ttime(tt) \leq tt_0$  do
           $t := tt$ 
           $tt := next\_record(tt)$ 
        od
      ;; if the record with closest transaction time is a  $C_1$ -corner, insert the corresponding item,
      ;; otherwise ignore
      if  $ttype(t) = INS$  then
        insert into timeslice ( $object\phi(t), item\phi(t)$ )
      fi
    fi
     $t := next\_record(t)$ 
  od
end

```

Decremental Transaction-Time Computation

```

procedure decrement-ttime( $TS_R(tt_0, vt_0), tt_D$ )
  use physical ordering of the backlog (index  $I_T$ )
  select last record  $t$  s.th.  $ttime(t) \leq tt_0$ 
  while  $ttime(t) > tt_D$  do
    while  $vtime(t) > vt_0$  do ;; ignore irrelevant records
       $t := \text{previous-record}(t)$ 
    od

    if  $ttype(t) = \text{INS}$  then ;; detect whether a validity rectangle is exited
      ;;  $t$  is the record with valid time closest to  $vt_0$  as the algorithm steps backwards through  $I_T$ 

      ;; if the record with closest valid time is a  $C_1$ -corner, delete the corresponding item,
      ;; otherwise ignore
      if  $vtype(t) = \text{BEGIN}$  then
        delete  $object\phi(t)$  from timeslice
      fi

    else ;;  $ttype(t) = \text{DEL}$ , detect whether a validity record is being entered
      ;;  $t$  is the record with valid time closest to  $vt_0$  as the algorithm steps backwards through  $I_T$ 

      ;; if the record with closest valid time is a  $C_3$ -corner, insert the corresponding item,
      ;; otherwise ignore
      if  $vtype(t) = \text{BEGIN}$  then
        insert into timeslice ( $object\phi, item\phi$ )
      fi

    fi
    ;; skip all remaining records that refer to the same object and have the same  $ttype$  as
    ;; the one just processed
     $tt := t$ 
    while  $object\phi(t) = object\phi(tt)$  and  $ttime(t) = ttime(tt)$ 
      and  $ttype(tt) = ttype(t)$  do
         $tt := \text{previous-record}(tt)$ 
      od
     $t := tt$ 
  od
end

```

Decremental Valid-Time Computation

```

procedure decrement-vtime( $TS_R(tt_0, vt_0), vt_D$ )
  use index  $I_V$ 
  select last record  $t$  s.th.  $vtime(t) \leq vt_0$ 
  while  $vtime(t) > vt_D$  do
    while  $ttime(t) > tt_0$  do ;; ignore irrelevant records
       $t := \text{previous-record}(t)$ 
    od

    if  $vtype(t) = \text{BEGIN}$  then ;; detect whether a validity rectangle is exited
      ;;  $t$  is the record with transaction time closest to  $tt_0$  as the algorithm steps backwards through
 $I_V$ 
      ;; if the record with closest transaction time is a  $C_1$ -corner, delete the corresponding item,
      ;; otherwise ignore
      if  $ttype(t) = \text{INS}$  then
        delete  $object\phi(t)$  from timeslice
      fi
    else ;;  $vtype(t) = \text{END}$ , detect whether a validity record is being entered
      ;;  $t$  is the record with transaction time closest to  $tt_0$  as the algorithm steps backwards through
 $I_V$ 
      ;; if the record with closest transaction time is a  $C_2$ -corner, insert the corresponding item,
      ;; otherwise ignore
      if  $ttype(t) = \text{INS}$  then
        insert into timeslice ( $object\phi(t), item\phi(t)$ )
      fi
    fi
    ;; skip all remaining records that refer to the same object and have the same  $vtype$  as
    ;; the one just processed
     $tt := t$ 
    while  $object\phi(t) = object\phi(tt)$  and  $vtime(t) = vtime(tt)$ 
      and  $vtype(tt) = vtype(t)$  do
         $tt := \text{previous-record}(tt)$ 
      od
     $t := tt$ 
  od
end

```

REFERENCES

- [1] M. Aleksic, M. Chamlee, L. Mark. "Differential Computation of Bitemporal Timeslices". Georgia Institute of Technology, Technical Report GIT-CC-96-26.
- [2] L. Bækgaard, L. Mark. "Incremental Computation of Time-Varying Query Expressions". *IEEE Trans. on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995.
- [3] R. Bayer, E. McCreight. "Organization and Maintenance of Large Ordered Indices". *Proc. 1970 ACM-SIGFIDET Workshop on Data Description and Access*, Houston, TX, Nov. 1970.
- [4] J. Ben-Zvi. "The Time Relational Model". Ph.D. thesis, Computer Science Department, UCLA, 1982.
- [5] J. A. Blakeley, P.-Å. Larson, F.W. Tompa. "Efficiently updating materialized views". *1986 Proc. SIGMOD Int'l Conf. Management of Data*, pp. 61-71, Washington, D.C., 1986.
- [6] C. Davies, B. Lazell, M. Hughes, L. Cooper. "Time is just another attribute – or at least, just another dimension.", *Proc. of the International Workshop on Temporal Databases, Zürich, Switzerland*, 17-18 September 1995, pp. 175-193.
- [7] H. Gunadhi, A. Segev. "Efficient Indexing Methods for Temporal Relations". *IEEE Transactions on Knowledge and Data Engineering* Vol. 5, No. 3, pp. 496-509, June 1993.
- [8] A. Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching". *Proc. ACM SIGMOD*, 1984.
- [9] C.S. Jensen, J. Clifford, S.K. Gadia, A. Segev, R. Snodgrass. "Glossary of temporal database concepts". *ACM SIGMOD Record*, vl. 21, no. 3, pp. 35-43, Sept. 1992.
- [10] C.S. Jensen, editor et al. "A Consensus Glossary of Temporal Database Concepts". *ACM SIGMOD Record*, Vol. 23, No. 1, pp 52-64, 1994.
- [11] C.S. Jensen, L. Mark, N. Roussopoulos. "Incremental Implementation Model for Relational Databases with Transaction Time". *IEEE Trans. on Knowledge and Data Engineering*, Vol. 3, No 4, pp 461-473, 1991.
- [12] C. S. Jensen, R. Snodgrass. "Temporal Specialization and Generalization", *IEEE*

Trans. on Knowledge and Data Engineering, Vol. 6, No. 6, December 1994.

- [13] C.S. Jensen, R.T. Snodgrass. "The Surrogate Data Type". *The TSQL2 Temporal Query Language*, Chapter 9. Kluwer Academic Publishers, 1995.
- [14] C.S. Jensen, R.T. Snodgrass, M.D. Soo. "The TSQL2 Data Model". *The TSQL2 Temporal Query Language*, Chapter 10. Kluwer Academic Publishers, 1995.
- [15] A. Kumar, V.J. Tsotras, C. Faloutsos. "Access Methods for Bi-Temporal Databases". *Proc. of the International Workshop on Temporal Databases, Zürich, Switzerland*, 17-18 September 1995, pp. 235-254, Springer-Verlag.
- [16] S. Lanka, E. Mays. "Fully Persistent B⁺ Trees". *Proc. ACM SIGMOD*, pp 426-435, 1991.
- [17] M.A. Nascimento, M.H. Dunham, R. Elmasri. "M-IVTT: A Practical Index for Bitemporal Databases". *DEXA '96, Zürich, Switzerland* (Springer's LNCS Vol 1134).
- [18] N. Roussopoulos. "An Incremental Access Method of ViewCache: Concept, Algorithms, and Cost Analysis". *ACM Transaction on Database Systems*, Vol. 16 No. 3, pp. 535-563, September 1991.
- [19] B. Schueler. "Update Reconsidered". *Architectures and Models in Data Base Management Systems*. Ed. G. M. Nijssen. North Holland Publishing Co., 1977.
- [20] R. Snodgrass. "Temporal Databases – Status and Research Directions". *SIGMOD Record*, Vol. 19, No. 4, December 1990
- [21] Tansel, Clifford, Gadia, Jajoda, Segev, Snodgrass. "Temporal Databases: Theory, Design and Implementation". Benjamin Cummings Publishing Co., 1993.
- [22] K. Torp, L. Mark, C.S. Jensen. "Efficient Timeslice Computation". to appear in *IEEE Trans. on Knowledge and Data Engineering*.
- [23] V.J. Tsotras, B. Gopinath, G. Hart. "Efficient Management of Time-Evolving Databases". *IEEE Trans. on Knowledge and Data Engineering*, Vol. 7, No 4, August 1995.

GLOSSARY

The greater part of this glossary is taken from [10]. The part taken from there will be enclosed in quotes without additional reference. Definitions from other sources are referenced separately. Text appearing outside the quotes gives new definitions and additional explanations where the thesis deviates from original definitions. As opposed to [10], this glossary is sorted alphabetically, not by topic.

Beginning

“The time-line supported by any temporal DBMS is, by necessity, finite and therefore has a smallest and largest representable chronon. The distinguished value *beginning* is a special valid-time instant preceding the smallest chronon on the valid-time line. Beginning has no transaction-time semantics.”

In this thesis, I will refer to valid time “0” instead of *beginning* to have a more compact representation.

Bitemporal Interval

“A *bitemporal interval* is a region in two-space of valid time and transaction time, with sides parallel to the axes. When associated in the database with some fact, it identifies when that fact, recording that something was true in reality during the specified interval of valid time, was logically in the database during the specified interval of transaction time.

A bitemporal interval can be represented with a non-empty set of bitemporal chronons.”

Bitemporal Relation

“A *bitemporal relation* is a relation with exactly one system supported valid time and exactly one system supported transaction time. As for valid-time relations and transaction-time relations, there are no restrictions as to how either of these temporal dimensions may be incorporated into the tuples.”

Bitemporal Timeslice Operator

The *bitemporal timeslice operator* combines the effect of the transaction- and valid-timeslice operators. In this thesis, I will only consider bitemporal timeslices that are valid at a certain instant in transaction time and valid time although the original definition of the transaction- and valid-timeslice operators allows temporal elements. The syntax used for bitemporal timeslices of a relation R is $TS_R(\text{ttime}, \text{vtime})$.

Chronon

“In a data model, a one-dimensional *chronon* is a non-decomposable time interval of some fixed, minimal duration. An n-dimensional chronon is a non-decomposable region in n-dimensional time. Important special types of chronons include valid-time, transaction-time and bitemporal chronons.”

Differential Computation

This term is used to mean either incremental or decremental computation. The same convention is used in [11].

Forever

“The distinguished value *forever* is a special valid-time instant following the largest chronon on the valid-time line. Forever has no transaction-time semantics.”

History

“A *history* is the temporal representation of an ‘object’ of the real world or of a database. Depending on the object, we can have *attribute histories*, *entity histories*, *relationship histories*, *schema histories*, *transaction histories*, etc.”

History-Timeslice Operator

The history-timeslice operator, applied to a bitemporal relation, returns either the transaction-time or valid-time history of all objects stored in the relation in a given time interval. The syntax used for history-timeslices of a relation R is $TS_R(ttbegin\ to\ ttend,\ vtime)$ or $TS_R(ttime,\ vtbegin\ to\ vtend)$.

Initiation

“The distinguished value *initiation*, associated with a relation, denotes the time instant when a relation was created. ‘Initiation’ is a value in the domain of transaction times and has no valid-time semantics.”

As for valid times, I will use the special value “0” to denote to *initiation*.

Instant

“An *instant* is a time point on an underlying time axis.”

Item

An item identifies a set of atomic attribute values. The state of an object at any bitemporal chronon can be described by the pair (object, item). It means that the attributes of “object” have values identified by “item”.

Object

An object in a temporal database corresponds to a tuple in a snapshot relation. It is an entity stored in the database with a time-dependent state called an item.

There is a one-to-n relationship between objects and items as each object can have many different states over time. However, during any bitemporal chronon, an object can have at most one item associated with it.

Pure-Timeslice Query

“A set of objects evolves over time and at each time instant an object may be added or deleted. The query is to find the set’s state (i.e., the objects contained in the set) as of some time t . [...] (Obviously, the time predicate can instead of a time instant be a time interval)[15].”

This definition is for transaction-time relations. In bitemporal relations, the time predicate is a bitemporal element.

Range-Timeslice Query

“The more general query [compared to the pure-timeslice query] is the range-timeslice problem, where the predicate includes a condition on the objects’ key space, i.e., find the objects that were “alive” at t and whose keys are in range k . (Obviously, the time predicate can instead of a time instant be a time interval)[15].”

This definition is for transaction-time relations. In bitemporal relations, the time predicate is a bitemporal element.

In general, such a query can be computed by first computing the corresponding pure-timeslice query and subsequently choosing all tuples from the result that are in range k .

Snapshot Relation

“Relations of a conventional relational database system incorporating neither valid-time nor transaction-time timestamps are *snapshot relations*.”

Temporal Element

“A *temporal element* is a finite union of n-dimensional time intervals. Special cases of temporal elements include *valid-time elements*, *transaction-time elements*, and *bitemporal elements*. They are finite unions of valid-time intervals, transaction-time intervals, and bitemporal intervals, respectively.”

Time Interval

“A *time interval* is the time between two instants. In a system that supports a time line composed of chronons, an interval may be represented by a set of contiguous chronons.”

Timestamp

“A *timestamp* is a time value associated with some object, e.g., an attribute value or a tuple. The concept may be specialized to valid timestamp, transaction timestamp, interval timestamp, instant timestamp, bitemporal-element timestamp, etc.”

Timestamp Granularity

“In the discrete model of time, the *timestamp granularity* is the size of each chronon in a timestamp interpretation. For example, if the timestamp granularity is one second, then the duration of each chronon in the timestamp interpretation is one second (and vice-versa).”

Timestamp Interpretation

“In the discrete model of time, the *timestamp interpretation* gives the meaning of each timestamp bit pattern in terms of some time-line clock chronon (or

group of chronons), that is, the time to which each bit pattern corresponds. The timestamp interpretation is a many-to-one function from time-line clock chronons to time-stamp bit patterns.”

Transaction Time

“A database fact is stored in a database at some point in time, and after it is stored, it is current until logically deleted. The *transaction time* of a database fact is the time when the fact is current in the database and may be retrieved. Transaction times are consistent with the serialization order of the transactions. Transaction-time values cannot be later than the current transaction time. Also, as it is impossible to change the past, transaction times cannot be changed. Transaction times may be implemented using transaction commit times, and are system-generated and supplied.”

Transaction-Time Relation

“A *transaction-time relation* is a relation with exactly one system-supported transaction time. As for valid-time relations, there are no restrictions as to how transaction times may be incorporated into the tuples.”

Transaction-Timeslice Operator

“The *transaction-timeslice operator* may be applied to any relation with transaction time timestamps. It takes as one argument the relation and as a second argument a transaction-time element whose greatest value must not exceed the current transaction time. It returns the argument relation reduced in the transaction-time dimension to just those times specified by the transaction-time argument.”

In this thesis, I will only use transaction time timeslices that are restricted to a single transaction time instant. A transaction time timeslice is then a special case of a history-timeslice of a relation R : $TS_R(\text{ttime}, 0 \text{ to Forever})$.

Transaction Time/Valid Time Grid

The transaction time/valid time grid is used to visualize data in a bitemporal relation. The vertical axis represents valid time. Along this axis one can see the history of the stored objects in valid time. The horizontal axis represents transaction time. Along this axis one can see the update activity in the relation.

This kind of representation is taken from [14].

Valid Time

“The *valid time* of a fact is the time when the fact is true in the modeled reality. A fact may have associated any number of instants and time intervals, with single instants and intervals being important special cases. Valid times are usually supplied by the user.”

Valid-Time Relation

“A *valid-time relation* is a relation with exactly one system supported valid time. There are no restrictions on how valid times may be incorporated into the tuples; e.g. the valid-times may be incorporated by including one or more additional valid-time attributes in the relation schema, or by including the valid-times as a component of the values of the application-specific attributes.”

Valid-Timeslice Operator

“The *valid-timeslice operator* may be applied to any relation with valid time timestamps. It takes as one argument the relation and as a second argument a valid-time element. It returns the argument relation reduced in the valid-time

dimension to just those times specified by the valid-time argument.”

In this thesis, I will only use valid time timeslices that are restricted to a single valid time instant. A valid time timeslice is then a special case of a history-timeslice of a relation R : $TS_R(0 \text{ to NOW}, \text{vtime})$.