Aalborg University
Department of Computer Science

Fredrik Bajersvej 7E ■ DK-9220 Aalborg Ø ■ Phone: +45 96 35 80 80

Title: Building a Temporal Cartridge for Oracle8*i*

Period: `period('01-02-2000', '12-06-2000');`

Project members:

Bo Gundersen

Kim Thrysøe

Supervisor:

Kristian Torp

Copies: 7

Pages: 69

Completed: June 12th 2000

Abstract:

A large number of applications manage time varying data most of them in an ad-hoc manner, with all temporal logic stored in the application layer. This is because none of the major ORDBMS vendors actively support management of temporal data as proposed by researchers.

This project aims to migrate parts of the large amount of research done on temporal databases into existing ORDBMS technology. The principal goals of the project is to examine two matters: One, if the task of managing temporal data in current ORDBMSs can be eased by extending it with custom ADTs and procedures. Two, to examine if such an extension can support efficient execution of known temporal queries. A practical approach is taken to these matters, and as a part of the answer an Oracle cartridge is designed and implemented.

The report is structured in three parts. The first concerns the reduction of code complexity. The second describe implementation specifics. Finally, the third part contains performance measurements and evaluations.

The result shows that although it is possible to obtain a substantial reduction in code complexity, the maturity of database extensibility features prevent efficient execution of temporal queries, thus limiting the usefulness of a temporal cartridge.

Aalborg Universitet
Insitut for Datalogi

Fredrik Bajersvej 7E ■ DK-9220 Aalborg Ø ■ Tlf: +45 96 35 80 80

Titel: Building a Temporal Cartridge for Oracle8*i*

Periode: `period('01-02-2000', '12-06-2000');`

Projektdeltagere:

Bo Gundersen

Kim Thrysøe

Projektvejleder:

Kristian Torp

Oplag: 7

Antal sider: 69

Dato: 12. juni 2000

Synopsis:

Mange applikationer håndterer data der varierer over tid, de fleste på en ad-hoc måde, hvor den temporale logik ligger i applikations laget. Dette er ikke overaskene, da ingen af de store OR-DBMS leverandører aktivt støtter håndteringen af temporal data.

Målet med dette projekt er, at gøre dele af den store mængde temporal forskning der er lavet tilgængelig på eksisterende ORDBMS teknologi. I projektet undersøger vi to ting, for det første om det er muligt at lette arbejdet med temporal data i de nuværende ORDBMSer ved hjælp af ADTer og bruger definerede funktioner. For det andet, om sådanne udvidelser understøtter effektiv eksekvering af temporale forspørgsler. I projektet er der taget en praktisk tilgangsvinkel til disse spørgsmål, og et Oracle cartridge er designet og implementeret.

Rapporten er indelt i tre dele, den første omhandler reduktion i kode kompleksiteten, den anden beskriver implementations detaljer, den sidste indeholder hastighedsmålinger og evaluering.

Resultaterne viser, at selvom det er muligt at opnå en stor reduktion i antal linier kode, så forhindrer det lave modenhendsniveau af OR-DBMS udvidelses teknologien effektiv udførelse af temporale forspørgsler, og dermed mindskes brugbarheden af et temporalt cartridge.

Preface

This report is the outcome of a master thesis project carried out at the Department of Computer Science at Aalborg University, Aalborg, Denmark.

The thematic frame for the thesis project is *database systems*. Within this frame, it is chosen to take a practical approach to the integration of support for time-varying data in existing database management systems.

The report is organized as follows. Chapter 1 contains an introduction to the initial problem of the project along with a definition of goals and requirements for the project. Chapter 1 also relates the work of developing a temporal cartridge to other topics from the temporal research community. Chapter 2 contains a discussion of how to reduce the expressional complexity of temporal queries. The chapter presents two example databases and identifies a number of interesting temporal query types, each of which are expressed for both example databases. The difference in query code complexity in the two databases is finally shown. Chapter 3 discusses concrete implementation considerations in relation to the extensibility features of the Oracle object-relational database. Chapter 4 presents the results of a performance test carried out for a subset of the features designed for the temporal cartridge. The test includes methods for indexing periods and the execution of the identified temporal query types. Finally, Chapter 5 concludes on the feasibility of designing a temporal cartridge. This evaluation is based on the goals and requirements defined in Chapter 1.

Aalborg, June 12th, 2000

Bo Gundersen

Kim Thrysøe

Contents

Chapter 1

Introduction

A large number of applications manage time-varying data. Often these applications do not take advantage of the large amount of research done in the area of temporal databases [RP92, WJW98]. Applications such as portfolio-management, financial applications, personnel administration and scheduling, e.g., travel booking, are prime examples of applications managing time-varying data [Sno00].

By definition databases store facts about a modeled world. Research suggests that two main time dimensions can be associated with these facts [Sno00, Jen99]: *valid time* and *transaction time.* The *valid time* associated with facts store information about when the fact is true in the modeled world. Valid time may span the past, present, and future and by definition all facts have a valid time, whereas it is not necessary that this valid time is recorded in the database. The database may have several valid times recorded for each fact. *Transaction time* records when the fact is current in the database. Transaction time cannot span into the future and may be interpreted as a subset of the valid time dimension. A database that records both valid time and transaction time is called bi-temporal [Je98]. A *user-defined time* dimension has also been suggested which has no known semantics to the database. Valid time is the most general time dimension in temporal databases and is the focus in this paper. This focus is based on the fact that ideas from valid-time support can be used to handle transaction time, user-defined time, and bi-temporal timestamped data.

In addition to fixed valid-time periods, it is also possible to specify growing periods. These are called now-relative periods, because their end time is the special temporal value *now*. *now* is not bound to a specific time value until it is accessed, where it evaluates to the time at that moment. A fixed period is for example "05-05-2000 - 05-06-2000" and a now relative one is "05-05-2000 - *now*." These periods will be equal when evaluated on June 5th, 2000.

Without temporal support from the DBMS, developers of temporal applications must express temporal queries in standard non-temporal query languages. This results in two problems, namely that of code which is (1) hard to understand by developers [Sno00], and (2) complex to execute by the DBMS [BSS97, TGJ99, MLI99]. As an example, a conventional join query can be written in three to four lines of SQL92, whereas a temporal join query may require as much as ten times as many lines of SQL92 [Sno00].

Although it must be expected that the work of the temporal database research community is gaining the interest of commercial database management system (DBMS) and database application vendors, the research results has not yet been integrated into any commercial products. If the research in temporal databases is to gain general public and commercial acceptance, it has to be available for use with the major object-relational DBMSs (ORDBMS). The ongoing work to include SQL/Temporal [Mel96] into the coming SQL:1999 standard is an effort to make it so [ME00].

In the light of the problem of transferring temporal support to DBMSs, we define a number of requirements and goals for the project. These requirements are the subject of the following section.

1.1 Goals and Requirements

The overall goals with and scope of the present work is as follows.

GOAL1: Examine the possibilities of easing the task of managing now-relative valid-time data in commercial ORDBMSs

GOAL2: Provide a framework in the form of an extensibility module for efficient execution of temporal statements.

The fact that most temporal research is related to relational data models and the widespread use of ORDBMSs in the industry is the reason that we focus on technologies and concepts that can be readily implemented on these platforms. Technologies include extensibility technologies such as the cartridge concept used by Oracle [RRM99], and the DataBlade concept used by Informix [DLM97].

Having defined the goals we now list five requirements for the temporal framework.

REQ1: Existing commercially available technology. The framework should use proven technologies, accepted by the industry, application developers, and major ORDBMS vendors.

REQ2: Simple code. The framework must make it easy to express temporal queries, in the fact that the number of lines of source code necessary and the complexity of it is reduced, compared to SQL92.

REQ3: Fast execution. Temporal queries that are expressed using the constructs of this work should execute at least as fast as temporal SQL92 based queries.

REQ4: Structural platform independence. The major components from which the frame-work is build, should be portable to major ORDBMSs.

REQ5: Horizontal support. The functionality of the system developed should cover a wide spectrum of temporal concepts and be generally useful.

REQ1 ensures that the framework can be used by developers with out much change. REQ2 ensures that the code written using augmented SQL[1] is less complex that the code written in SQL92. REQ3 ensures that the performance of the augmented SQL should be at least as fast as the SQL92 code. REQ4 ensures that although the implementation is done in the Oracle ORDBMS, the design is portable to other ORDBMSs. REQ5 ensures that the framework developed can be used to express a broad range of temporal queries.

Because of REQ4, REQ5, and REQ1, Oracle's cartridge technology is chosen as the basis for implementing the framework.

1.2 Related work

When adding temporal support to any system, four different approaches can be taken [Böh95]. Each of these approaches has its advantages and disadvantages. The approaches are as follows.

1. Application. In the application approach, the application itself has the responsibility of handling the temporal semantics. This is done on top of a conventional DBMS [Sno00].

[1]Augmented SQL is ORDBMS vendor specific SQL implementations containing object relational constructs referencing the functionality of the temporal cartridge developed in this project.

2. Layer. In the layered approach, systems implement a layer between the application and the DBMS. The layer translates from a temporal query language such as TSQL2 [Sno95] to standard SQL. This approach is described in [TJS98].

3. ORDBMS. It is also possible to embed some temporal extensions in an ORDBMS, using extensibility interfaces. Not much research has been done in this area, but a concrete example of a project that uses the embedding approach is TIP [YWY99], a temporal object-model for Informix. This is also the approach taken in this project.

4. Core. An approach is to implement the handling of temporal semantics in the core of the DBMS. In this approach applications use a temporal query language to query the database directly. This is the approach taken in some versions of the Postgres DBMS [RS87].

Various books [Sno95, Sno00, Jen99] and articles [BBS98, DSJ93, DS91] cover the semantics of temporal data. Query languages have been suggested [BSS97, YC91, Sno95, BJ96] including initiatives to add temporal support to the SQL:1999 standard [Mel96, ME00].

Several structures for indexing temporal data has been suggested, including the use of B^+-trees [ND98], GR-trees [BSSJ98], MVB trees [dBS96], and R-tree based structures [BJSS98, SN98].

Temporal algorithms include coalescing [BSS97], difference [TGJ99], aggregation [KS95, MLI99], time-slice [TJS98], and join [PJ98].

Temporal concepts have been implemented in various prototype database systems. Tiger [BBM+99] is an implementation based on the temporal query language ATSQL [BJ96]. TimeDB [BJSS95] is a similar approach, also based on the temporal query language ATSQL. TIP [YWY99] uses a different method, which is very like the one taken in the work of this paper, namely to add temporal data types to an existing ORDBMS using object-relational extensibility features. In such an approach queries are expressed using user-specified operators. This is the subject of the next chapter.

Chapter 2

Expressing Temporal Queries

In this section we present a framework that makes temporal queries easier to express. The section is aimed at REQ1 and REQ4, and is structured as follows. First basic temporal query types are identified, then two database designs in a running example from a fictitious tele communication company called TerraTele is introduced. The TerraTele example serves as an illustration in the sections to come. The first database (which we will refer to as *conventional*) is designed with temporal support as described in the literature [Sno00] while the other (called *augmented*) utilizes new temporal data types provided by the temporal cartridge designed in the project.

For each of the temporal query types of interest, we describe the query and point out what for and where the query may be used. Following this simple description more thorough examples of the temporal query, based on the conventional and augmented TerraTele database, is discussed.

At the end of the section we evaluate the improvements possible with the augmented SQL. The evaluation is focused on the reduction in complexity of the queries, and is based on reduction in lines of code.

The temporal type system later described (Chapter 3) supports now relative data. This is not the case with the SQL92 queries in the following sections, which are expressed to work on non-now-relative data only. The new, augmented queries in this chapter can thus support now-relative data, as this fact does not change the way those queries are expressed.

2.1 Temporal Query Types

Temporal queries can be divided into three kinds: Current/time-slice queries, sequenced queries, and non-sequenced queries [BSS97]. Sequenced queries are the most complex of the three to express [Sno00] and are the subject of this investigation. A sequenced query can be viewed as a conventional query executed sequentially at each of the states of a temporal relation. Non-sequenced queries make no use of the fact that timestamps associated with data have special semantics. This argues why we are not interested in exploring non-sequenced queries.

Each of the relational operators *selection*, *projection*, *join*, *difference*, *union*, *intersection*, and *aggregation* [SKS97] has a temporal counterpart. Furthermore two special temporal relational operators exists, namely the *coalesce* and the *time-slice* operators [Je98].

Sequenced selection and projection are not considered as they are simple to express in the fact that they are similar to the snapshot counterparts, except that they also reference the two extra attributes. This leaves us with the *coalescing* and *time-slice* operators and the following sequenced operators to consider in the present work : *join*, *difference*, *union*, *intersection*, and *aggregation*.

In order to express examples of these queries we present two example databases.

2.2 Schema for TerraTele

TerraTele is a fictitious tele communication company for which the two databases are designed. The databases cover the same modeled world, namely how persons subscribe to services and place telephone calls. The one database is designed only with data types available in todays RDBMSs. This design is illustrated in the ER diagram in Figure 2.1. The other database is designed using the augmented temporal data types (for an introduction to the new types see Section 3.1.1). The second design is shown in the ER diagram in Figure 2.2.

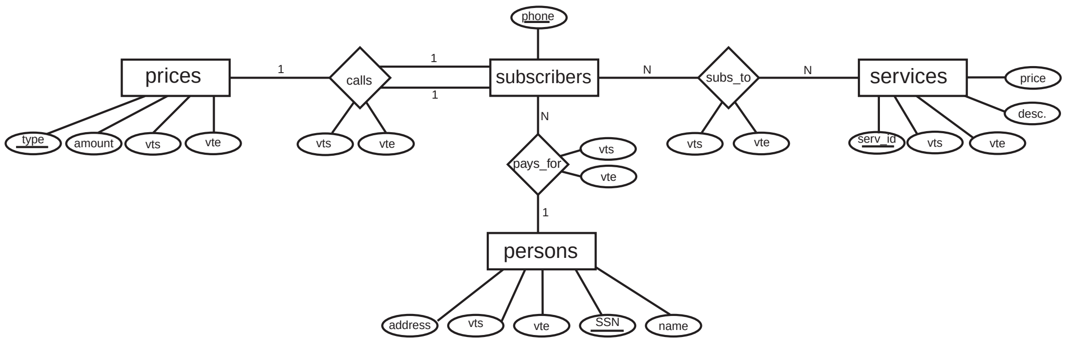

Figure 2.1: *ER Diagram Showing the Conventional TerraTele Database Design*

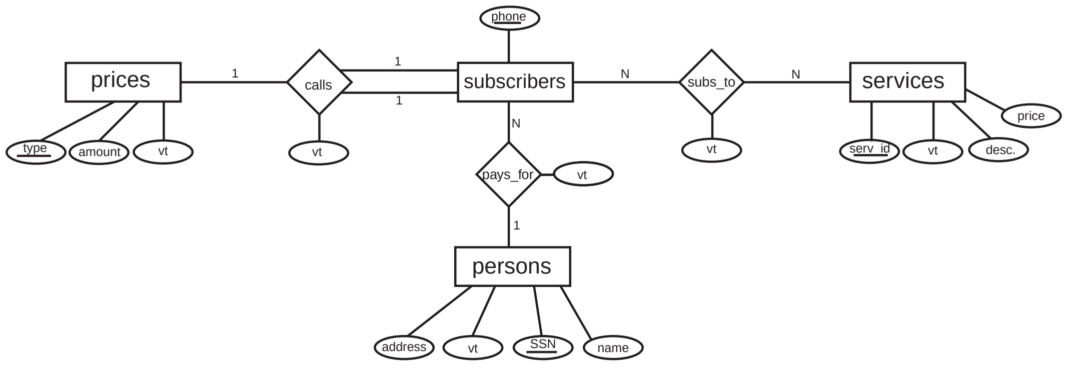

Figure 2.2: *ER Diagram Showing the TerraTele Database Design Using Augmented Data Types*

The temporal ER model shown in Figure 2.2 is not an example of temporal ER modeling, but is an attempt to stay as close to the conventional ER model as possible, while still optimizing temporal queries. This is done to ensure compatibility with REQ1. For a description of temporal conceptual modeling, see [GJ97].

Each entity in both schemas is defined with a primary key, but because the databases contain temporal data, the issue of primary keys are non-trivial. The semantics of the primary keys used are that of temporal primary keys, and is described in [Sno00]. The functionality of temporal primary keys are not implemented.

The conventional TerraTele Database is explained next, followed by a discussion of how the enhanced database differs.

Conventional Database Design

The ER model for the conventional TerraTele database contains the following four entities

Entity	Attribute	Description
subscribers	phone	Subscriber telephone number.
prices	type	Description of the price type.
	amount	The amount of money the price category costs.
	vts	The date describing when a price category came into effect.
	vte	The date describing when the price category no longer is in effect.
services	serv_id	Unique service id.
	description	A textual description of the service.
	price	The price of the service.
	vts	Point in time from which the service was available.
	vte	Point in time from which the service was no longer available.
persons	SSN	The persons social security number.
	name	The persons name.
	address	The persons address.
	vts	The time from which this person was a customer and registered in the database.
	vte	The time from which this person was no longer a customer.

The entities are related by three relationships as follows.

calls The terteriary *calls* relationship relates two subscribers and a price with each other to form a *telephone call.* Each subscriber can be associated with any number of other subscribers and prices, but never with more than one at any point in time.

pays_for *Pays_for* relates a subscription to specific persons. At each point in time, each person may appear as several subscribers whereas each subscriber is associated with exactly one person.

subs_to *Subs_to* relates subscribers to the services they subscribe to. A subscriber may subscribe to any number of services and a service can be subscribed by any number of subscribers.

The ER model gives rise to the seven tables listed in the table below.

Table	Attribute	Description
subscribers	phone	The phone number associated with this subscription.
pays_for	phone	Telephone number.
	SSN	Subscribers association with a person
	vts	The time from when a person is associated with the subscriber.
	vte	The time from when a person is no longer associated with the subscriber.
prices	type	The type of price, can for example be "international call."
	amount	The price of this type.
	vts	The valid time start of this type of price, i.e., from when this price was effective.
	vte	The valid time end, i.e. from when this price was no longer valid.
calls	caller	The calling subscriber.
	callee	The subscriber that receives the call.
	vts	The start time of the call.
	vte	The end time of the call.
persons	SSN	The social security number of the person.
	name	The name of a person.
	address	The address of the person.
	vts	The time from when this person was valid.
	vte	The time from when this person was no longer valid.
subs_to	phone	Foreign key to subscriber. The subscriber involved in the subscription.
	serv_id	Foreign key to services. The service involved in the subscription.
	vts	The time from when this subscriber subscribed to this service.
	vte	The from when this subscriber no longer subscribed to this service.
services	serv_id	The service identifier.
	desc	Textual description of the service.
	price	The price of this service measured in amount per month.
	vts	From when this service was valid.
	vte	To when this service was valid.

Enhanced Database Design

The enhanced database differs from the description above, in the fact that the timestamps have been replaced with the data types specified in the present work. The *vts* and *vte* timestamps have been changed in this way to *Period* attributes in the following entities and relationships: *prices*, *persons*, *services*, *calls*, *subs_to*, and *is_a*. The start and end points can be accessed as *vt.s* and *vt.e*.

Modification of the Database

When working with temporal data, insert, delete and update operations are performed differently than when working with non-temporal data.

Inserts into a valid-time table can be done in two ways, either the tuple has a specified valid-time or it is assigned a valid time. When inserting tuples with a specified valid-time, it is possible to insert tuples that was valid in the past, or is valid in the future. The standard valid-time assigned to newly inserted tuples are however from the current time till *now*.

When deleting tuples from a valid-time table, the tuple is not physical deleted. Instead the tuples "valid-time end" is changed from *now*, to the current time.

A temporal update is, much like a non-temporal update, conceptually a temporal deletion followed by a temporal insertion.

The semantics described above is a simplified version of the temporal semantics described in [BJ97], which cover the modification of temporal data in more detail.

Inserting, updating, and deleting in the augmented database is performed in much the same way as in the conventional temporal database. Instead of updating the individual timestamps, the period object is changed accordingly.

2.3 Temporal Queries

This section contains a description of the *join*, *set*, *coalescing*, *time-slice*, and *aggregation* operators.

2.3.1 Expressing Join

A join query combines information from two or more tables. In relational database design, information is split between tables by the normal forms [SKS97]. When querying the database for information, this distribution of data often results in the fact that the data wanted is stored in several tables. Therefore the query has to combine this information.

Temporal joins have added complexity compared to non-temporal joins. In a non-temporal join, tuples from each table is compared only on the join predicate. In valid time temporal joins, they are also compared on valid time and only tuples with overlapping valid time are added to the result. The valid time of the resulting tuples, is computed as the intersection of the two source tuples. For a formal description of the semantics of temporal joins see [BJ97].

SQL92

An example of a valid time temporal join query is written in Code Listing 1. The SQL92 code in Listing 1 returns the price of each call that subscriber **X** has made, in the *Period* **Y**. It combines information from three tables, namely *subscribers*, *calls* and *prices*.

Expressing a valid time temporal join in standard SQL92, is written as four separate SELECT statements unioned together. The query is split into four parts based on how the valid time of the two source tuples can relate to each other, and therefore what should be the valid time of the resulting tuple. Two such periods can relate to each other in six different ways, two of which are not interesting for join because they do not overlap. Figure 2.3 shows the remaining four ways a call can overlap a price. Lines 1 to 10 of Code Listing 1 matches case 1, lines 11 to 21 matches case 3, lines 22 to 31 matches case 2, and finally lines 33 to 41 matches case 4.

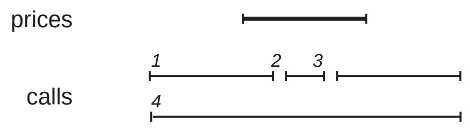

Figure 2.3: *How the Valid Time of a Call can Overlap the Valid Time of a Price.*

Code Listing 1 - *Temporal Join Written in SQL92*

```
/* Call left overlaps Prices */
SELECT s.phone, c.callee, (c.VTE-p.VTS)*p.amount
FROM Prices p, Calls c, Subscribers s
WHERE c.VTS < p.VTS AND
      c.VTE > p.VTS AND
      c.VTE < p.VTE AND
      c.VTS >= Y.VTS AND
      c.VTS <= Y.VTE AND
      s.phone = X AND
      c.caller = s.phone
UNION ALL
/* Call right overlaps Prices */
SELECT s.phone, c.callee, (p.VTE-c.VTS)*p.amount
FROM Prices p, Calls c, Subscribers s
WHERE c.VTS > p.VTS AND
      c.VTS < p.VTE AND
      c.VTE > p.VTE AND
      c.VTS >= Y.VTS AND
      c.VTS <= Y.VTE AND
      s.phone = X AND
      c.caller = s.phone
UNION ALL
/* Call is within Prices */
SELECT s.phone, c.callee, (c.VTE-c.VTS)*p.amount
FROM Prices p, Calls c, Subscribers s
WHERE c.VTS > p.VTS AND
      c.VTE < p.VTE AND
      c.VTS >= Y.VTS AND
      c.VTS <= Y.VTE AND
      s.phone = X AND
      c.caller = s.phone
UNION ALL
/* Call contains Prices */
SELECT s.phone, c.callee, (p.VTE-p.VTS)*p.amount
FROM Prices p, Calls c, Subscribers s
WHERE c.VTS < p.VTS AND
      c.VTE > p.VTE AND
      c.VTS >= Y.VTS AND
      c.VTS <= Y.VTE AND
      s.phone = X AND
      c.caller = s.phone
```

As we can see from Code Listing 1, each of the four parts returns a different valid time. This is so because only the intersection of the two tuples serve as the valid time of the result tuple.

Augmented SQL

As described the reason for splitting the query into four parts, was to return the intersection of the valid time of the two source tuples. This can be done with the *Intersect* method on the *Period* object. The *Intersect* methods returns a new *Period* object, which is the intersection between the two input *periods*.

Another part of the query is to make sure that we only consider overlapping tuples, this can be done by using the *Overlaps* method as a predicate for the SELECT statement.

By using the *Intersect* and *Overlaps* methods, the join query from Code Listing 1 can be expressed as shown in Code Listing 2

Code Listing 2 - *Temporal join written in augmented SQL*

```
SELECT s.phone, c.callee, c.vt.Intersect(p.vt)*p.amount
FROM Subscribers s, Calls c, Prices p
WHERE c.vt.Overlaps(p.vt) = 1 AND
      c.vt.Overlaps(Y) AND
      s.phone = X AND
      c.caller = s.phone
```

2.3.2 Expressing Set Operations

Applying set operations (i.e. union, intersection, and difference) on temporal data is different from the case of non-temporal data. The reason for this is that when expressing the temporal query it must be taken into account, that the valid time of a period must be inspected and most often will be changed for the result. Temporal union can be expressed as a query that either eliminates or retains temporal duplicates [BSS97] in the result. The version where duplicates are retained is trivial, as it is simply expressed in the same way as a snapshot union. Temporal union where duplicates are eliminated correspond to coalescing the result of a snapshot union, and is therefore also, by itself, trivial to express in SQL92. The intersection operator may be expressed as either two set differences or as a sequenced equi-join with the equality predicate covering all attributes.

Set difference is conceptually quite simple, but difficult to express. The concept is to subtract periods of tuples with matching explicit attributes whose periods overlap or are adjacent. Figure 2.4 illustrates this, by showing one tuple from the *services* table and two tuples from the *subs_to* table. The bottom line shows the two tuples that result from subtracting the *subs_to* tuples from the *services* table.

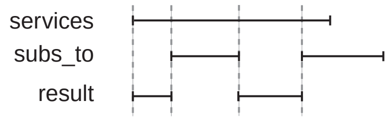

Figure 2.4: *Illustration of the Set Difference Operator*

Concretely consider the following two examples of the *services* and *subs_to* tables respectively:

services				
serv_id	**desc.**	**price**	**vts**	**vte**
MOBILE	Mobile service	10	2	12
LONGDIS	Long distance call	20	2	10
LONGDIS	Long distance call	30	11	15

In this table we see two services, MOBILE and LONGDIS. MOBILE costs 10 from time 2 to 12, while LONGDIS costs 20 from time 2 to time 10 and then 30 from time 11 to 15. The table is coalesced and therefore has no temporal duplicates.

subs_to			
phone	**serv_id**	**vts**	**vte**
555-1	LONGDIS	5	10
555-2	MOBILE	2	4
555-3	MOBILE	2	3
555-3	MOBILE	5	9

The *subs_to* table is also coalesced and contains the mapping from subscribers to services. In this way we can see that 555-1 subscribed to LONGDIS from time 5 to 10. Subscriber 555-2 subscribes to the MOBILE service from time 2 to 4, and the last subscriber, 555-3, subscribes to mobile from 2 to 3 and again from time 5 to 9.

The two tables are not union compatible, which they have to be in order to use them in relation with the set difference operator. After projecting the tables, the result of performing a set difference operation on the *subs_to* table and the *services* table yields the following result.

services \ *subs_to*		
serv_id	**vts**	**vte**
MOBILE	10	12
LONGDIS	2	4
LONGDIS	11	15

This result means that nobody subscribed to the MOBILE service from time 10 to 12. From time 2 to 4 the LONGDIS was not subscribed to which was also the case from time 11 to 15.

For a description of the formal semantics of set difference, union, and intersection see [BJ97].

SQL92

Expressing set difference in SQL92 can be written as a four part statement [Sno00], an example of such a statement can be seen in Code Listing 3.

The four sub-queries represent the four ways that an output row can be found. The first case, in lines 1 to 7, is where a service is never subscribed to, so the entire period is returned. In the second case, in lines 9 to 18, the service starts to exist before a subscriber begins a subscription, i.e., the *subs_to* period overlaps the *services* period to the right. The output tuple will in this case have a period that goes from the services period start to the start time of the subscription. In the third case, shown in lines 20 to 29, the subscription was for some reason terminated before the service ceased to exist. The resulting tuple will then go from the subscription end time to the service end time. The fourth and last case, shown in lines 31 to 45, handles holes in the subscription period. An example of such a hole, is the second range of the result in Figure 2.4

Code Listing 3 - *Set Difference Written in SQL92*

```
SELECT p1.serv_id , p1.desc , p1.VTS, p1.VTE
FROM services p1
WHERE NOT EXISTS (SELECT *
                  FROM subs_to s3
                  WHERE p1.serv_id = s3.serv_id AND
                        p1.VTS < s3.VTE AND
                        s3.VTS < p1.VTE)
UNION ALL
SELECT p1.serv_id , p1.desc , p1.VTS, s1.VTS
FROM services p1, subs_to s1
WHERE p1.serv_id = s1.serv_id AND
            p1.VTS < s1.VTS AND
            s1.VTS < p1.VTE AND
            NOT EXISTS (SELECT *
                        FROM  subs_to s3
                        WHERE p1.serv_id = s3.serv_id AND
                              p1.VTS < s3.VTE AND
                              s3.VTS < s1.VTS)
UNION ALL
SELECT p1.serv_id , p1.desc , s1.VTE, p1.VTE
FROM services p1, subs_to s1
WHERE p1.serv_id = s1.serv_id AND
            s1.VTE < p1.VTE AND
            p1.VTS < s1.VTE AND
            NOT EXISTS (SELECT *
                        FROM subs_to s3
                        WHERE p1.serv_id = s3.serv_id AND
                              s1.VTE < s3.VTE AND
                              s3.VTS < p1.VTE)
UNION ALL
SELECT p1.serv_id , p1.desc , s1.VTE, s2.VTS
FROM services p1,subs_to s1, subs_to s2
WHERE p1.serv_id = s1.serv_id AND
          s2.serv_id = s1.serv_id AND
          s2.phone = s1.phone AND
          s1.VTE < s2.VTS AND
          p1.VTS < s1.VTE AND
          s1.VTS < p1.VTE AND
          p1.VTS < s2.VTE AND
          s2.VTS < p1.VTE AND
            NOT EXISTS (SELECT *
                        FROM subs_to s3
                        WHERE p1.serv_id = s3.serv_id AND
                              s1.VTE < s3.VTE AND
                              s3.VTS < s2.VTS)
```

Augmented SQL

Code Listing 4, shows the query expressed using augmented SQL.

Code Listing 4 - *Set Difference Written in Augmented SQL*

```
SELECT p1.serv_id, p1.desc, create_period(p1.vt.s, p1.vt.e)
FROM services p1
WHERE NOT EXISTS (SELECT *
                  FROM subs_to s3
                  WHERE p1.serv_id = s3.serv_id AND
                        p1.vt.overlaps(s3.vt) = 1)
UNION ALL
SELECT p1.serv_id, p1.desc, create_period(p1.vt.s, s1.vt.s)
FROM services p1, subs_to s1
WHERE p1.serv_id = s1.serv_id AND
            s1.vt.StartsInside(p1.vt) = 1 AND
            NOT EXISTS (SELECT *
                        FROM  subs_to s3
                        WHERE p1.serv_id = s3.serv_id AND
                              s3.vt.overlaps(create_period(p1.vt.s, s1.vt.s)) = 1)
UNION ALL
SELECT p1.serv_id, p1.desc, create_period(s1.vt.e, p1.vt.e)
FROM services p1, subs_to s1
WHERE p1.serv_id = s1.serv_id AND
            s1.vt.EndsIndside(p1.vt) = 1 AND
            NOT EXISTS (SELECT *
                        FROM subs_to s3
                        WHERE p1.serv_id = s3.serv_id AND
                              s3.vt.overlaps(create_period(s1.vt.e, pe.vt.e)) = 1)
UNION ALL
SELECT p1.serv_id, p1.desc, create_period(s1.VTE, s2.VTS)
FROM services p1,subs_to s1, subs_to s2
WHERE p1.serv_id = s1.serv_id AND
          s2.serv_id = s1.serv_id AND
          s2.phone = s1.phone AND
          s1.vt.overlaps(p1.vt) = 1 AND
          s2.vt.overlaps(p1.vt) = 1 AND
          s1.vt.e < s2.vt.s AND
          NOT EXISTS (SELECT *
                      FROM subs_to s3
                      WHERE p1.serv_id = s3.serv_id AND
                            s3.vt.overlaps(create_period(s1.vt.e, s2.vt.s)) = 1)
```

The overall structure of the augmented version is the same as the standard SQL92 version. This is because we still have to distinguish between the different ways of overlapping in order to handle the special case of periods being split in two.

The *create_period* method is a function used as a constructor of periods. *StartInside* and *EndInside* are functions that specify whether a period in question starts or ends inside another given period. For the semantics of these functions see Appendix A.

2.3.3 Expressing Coalescing

Coalescing temporal data is similar in concept to removing duplicates from conventional data. The concept being that tuples in a table with matching explicit attributes, and overlapping or adjacent valid-times contains duplicate information. When coalescing a table, tuples with matching explicit attributes, and overlapping or adjacent valid-times are combined into one tuple with a valid-time that is the union of the source tuple valid-times.

As an example consider the following table, it is a small part of the *subs_to* table.

subs_to			
phone	**serv_id**	**vts**	**vte**
555-1	LONGDIS	1	4
555-1	LONGDIS	5	10
555-2	MOBILE	1	5
555-2	MOBILE	4	8
555-3	MOBILE	1	3
555-3	MOBILE	5	9

In the table we can see that 555-1 subscribed to the LONGDIS product from 1 to 4 and again from 5 to 10, 555-2 subscribed to the MOBILE product from 1 to 5 and from 4 to 8, and finally that 555-3 subscribed to the MOBILE product from 1 to 3 and from 5 to 9. The table contains two kinds of uncoalesced data, first 555-1's subscription to LONGDIS from 1 to 4 and again from 5 to 10 is adjacent. Also 555-2's subscription to MOBILE is uncoalesced because of the overlap of periods 1 to 5 and 4 to 8.

If we coalesce the table, the result is as follows.

coalesced *subs_to*			
phone	**serv_id**	**vts**	**vte**
555-1	LONGDIS	1	10
555-2	MOBILE	1	8
555-3	MOBILE	1	3
555-3	MOBILE	5	9

For a detailed description of the formal semantics of coalescing see [BJ97]

SQL92

Expressing coalescing in SQL92 can be written as a three part statement [BSS97], an example of such a statement can be seen in Code Listing 5.

Code Listing 5 - *Coalescing Written in SQL92*

```
SELECT DISTINCT f.phone, f.serv_id, f.vts, l.vte
FROM subs_to f, subs_to l
WHERE f.vts < l.vte AND
      f.phone = l.phone AND
      f.serv_id = l.serv_id AND
      NOT EXISTS (SELECT *
                  FROM subs_to m
                  WHERE f.phone = m.phone AND
                        f.serv_id = m.serv_id AND
                        f.vts < m.vts AND
                        m.vts < l.vte AND
                        NOT EXISTS (SELECT *
                                    FROM subs_to a1
                                    WHERE f.phone = a1.phone AND
                                          f.serv_id = a1.serv_id AND
                                          a1.vts < m.vts AND
                                          m.vts <= a1.vte)) AND
NOT EXISTS (SELECT *
            FROM subs_to a2
            WHERE f.phone = a2.phone AND
                  f.serv_id = a2.serv_id AND
                  (a2.vts < f.vts AND f.vts <= a2.vte OR
                   a2.vts <= l.vte AND l.vte < a2.vte))
```

The first part (lines 1 to 5) selects two value-equivalent tuples, and uses them as start and end points of the resulting tuple. The second part (lines 6 to 17) ensures that a chain of value-equivalent tuples cover the entire valid-time between the start and end points selected in the first part. The last part (lines 18 to 23) ensures that the start and end points selected in the first part, cover the longest possible period.

Augmented SQL

Code Listing 6, shows the query expressed using augmented SQL.

Code Listing 6 - *Coalescing Written in Augmented SQL*

```
SELECT DISTINCT f.phone, f.serv_id, Create_period(f.vt.s, l.vt.e)
FROM subs_to f, subs_to l
WHERE f.vt.s < l.vt.e AND
      f.phone = l.phone AND
      f.serv_id = l.serv_id AND
      NOT EXISTS (SELECT *
                  FROM subs_to m
                  WHERE f.phone = m.phone AND
                        f.serv_id = m.serv_id AND
                        m.vt.LeftOverlap(f.vt.s, l.vt.e) AND
                        NOT EXISTS (SELECT *
                                    FROM subs_to a1
                                    WHERE f.phone = a1.phone AND
                                          f.serv_id = a1.serv_id AND
                                          a1.vt.LeftOverlap(m.vt) = 1)) AND
NOT EXISTS (SELECT *
            FROM subs_to a2
            WHERE f.phone = a2.phone AND
                  f.serv_id = a2.serv_id AND
                  (a2.vt.s < f.vt.s AND f.vt.s <= a2.vt.e OR
                   a2.vt.s <= l.vt.e AND l.vt.e < a2.vt.e))
```

The structure of the augmented version is the same as the standard SQL92 version, and the augmentations is not used much. The size of the augmented version is 2 lines smaller than the standard version. The *LeftOverlaps* function takes a period and returns true if it overlaps the end point of the *period* it is compared with.

2.3.4 Expressing Time-slice

The time-slice query is a temporal query, used to slice the data in the database along a time-dimension, thereby viewing the data stored in the database at that time (transaction-time) or how the modeled world looked at that time (valid-time).

An example of a time-slice query is to find the calls that where ongoing at a given time. The following table is an example of the data in the *calls* table.

calls

caller	**callee**	**vts**	**vte**
555-1	555-2	1	10
555-3	555-4	2	4
555-3	555-5	5	7

From this table we can see that 555-1 called 555-2 from 1 to 10, and 555-3 called 555-4 from 2 to 4 and 555-1 from 5 to 7. If we time-slice the table at 6, we get the following table.

calls time sliced	
caller	**callee**
555-1	555-2
555-3	555-5

SQL92

The expression of a time-slice query in SQL92 is very straight forward, as seen in Code Listing 7. This query time-slices the *calls* at the time point **X**.

Code Listing 7 - *Time-slice Written in SQL92*

```
SELECT caller, callee
FROM Calls
WHERE vts <= X AND
      vte >= X;
```

Augmented SQL

As with SQL92 it is straight forward to express time-slice in augmented SQL, the only difference being that the predicate is changed to use an *Overlaps* method. The code for augmented SQL time-slice can be seen in Code Listing 8.

Code Listing 8 - *Time-slice Written in Augmented SQL*

```
SELECT caller, callee
FROM Calls
WHERE vt.Overlaps(X) = 1;
```

2.3.5 Expressing Aggregation

Aggregation queries summarizes data, and presents them in a more compact and informative way. They can be simple as counting the number of employees or calculating the average salary in the R&D department, or complex like showing the development in the number of customers over time. The latter is an example of a temporal aggregation query, that summarizes over time.

In the example from Section 2.2 the data from the *persons* table can be used to count the number of customers related to the company at any given time. The following table shows an example of the date contained in the persons table.

persons				
SSN	**name**	**address**	**vts**	**vte**
1	John	Wall Street	1	10
2	Jane	Yonge Street	1	3
3	Joe	El Camino Real	5	11

From the table we can see that John was a customer from 1 to 10, Jane from 1 to 3 and Joe from 5 to 11. Using this data to calculate the number of customers related to the company would yield the following result.

aggregated *persons*		
count	**from**	**to**
2	1	3
1	4	4
2	5	10
1	11	11

Figure 2.5 illustrates how the the aggregation result is found.

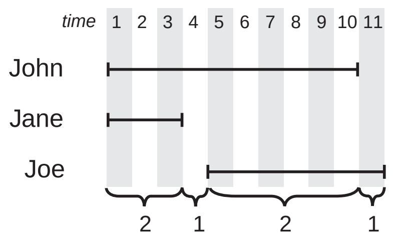

Figure 2.5: *Aggregation of the Persons Table*

SQL92

When expressing temporal aggregations, one very important part of the query is finding constant regions. That is, regions where the information being aggregated did not change. As shown in Code Listings 9 and 10 the SQL code from line 3 in Listing 9 to line 52 in Listing 10 is responsible for finding constant regions.

When the constant regions are found, a count is made for each constant region.

Augmented SQL

Although it is not possible to make generic table operators that is schema independent, it is possible to make functions with a table operator like functionality with certain limitations [Thr00]. It is possible to make a function that, given a table name and a *Period* column name, can return the constant regions of that table. By using this function, the code shown in Code Listing 9 and 10 can be expressed as shown in Code Listing 11.

2.4 Evaluation of Code Complexity

The idea with this Chapter is to evaluate the possibility of fulfilling REQ2 under the restraints imposed by the implementation environment. We have shown examples of the most common queries, expressed both in SQL92 and in the augmented SQL proposed in this work.

To evaluate on the complexity of these queries, we compare the number of lines of code necessary to express the query in SQL92 and augmented SQL respectively. The following table contains a list of queries and the number of lines of code for both SQL92 and augmented SQL.

Code Listing 9 - *Aggregation Written in SQL92 (part 1 of 2)*

```
SELECT COUNT(Persons.SSN), agg_table.vts AS vts, agg_table.vte AS vte
FROM Persons, (
  /* No start or stop overlap of p1 */
  SELECT p1.vts AS vts, p1.vte AS vte
  FROM Persons p1
  WHERE NOT EXISTS (SELECT *
                    FROM Persons p2
                    WHERE ((p1.vts < p2.vts AND
                            p2.vts < p1.vte)
                           OR
                            (p1.vts < p2.vte AND
                             p2.vte < p1.vte)))
  UNION
  /* Gap from p1.vte to p2.vts */
  SELECT p1.vte AS vts, p2.vts AS vte
  FROM Persons p1, Persons p2
  WHERE p1.vte < p2.vts AND
        NOT EXISTS (SELECT *
                    FROM Persons p3
                    WHERE ((p1.vte < p3.vts AND
                            p3.vts < p2.vts)
                           OR
                            (p1.vte < p3.vte AND
                             p3.vte < p2.vts)))
  UNION
  /* p2 left overlaps p1: First */
  SELECT p2.vts AS vts, p1.vts AS vte
  FROM Persons p1, Persons p2
  WHERE p2.vts < p1.vts AND
        p1.vts < p2.vte AND
        p2.vte < p1.vte AND
        NOT EXISTS (SELECT *
                    FROM Persons p3
                    WHERE ((p2.vts < p3.vts AND
                            p3.vts < p1.vts)
                           OR
                            (p2.vts < p3.vte AND
                             p3.vte < p1.vts)))
  UNION
  /* p2 left overlaps p1: Second */
  SELECT p1.vts AS vts, p2.vte AS vte
  FROM Persons p1, Persons p2
  WHERE p2.vts < p1.vts AND
        p1.vts < p2.vte AND
        p2.vte < p1.vte AND
        NOT EXISTS (SELECT *
                    FROM Persons p3
                    WHERE ((p1.vts < p3.vts AND
                            p3.vts < p2.vte)
                           OR
                            (p1.vts < p3.vte AND
                             p3.vte < p2.vte)))
  UNION
```

Code Listing 10 - *Aggregation Written in SQL92 (part 2 of 2)*

```
    /* p2 left overlaps p1: Third */
    SELECT p2.vte AS vts, p1.vte AS vte
    FROM Persons p1, Persons p2
    WHERE p2.vts < p1.vts AND
          p1.vts < p2.vte AND
          p2.vte < p1.vte AND
          NOT EXISTS (SELECT *
                      FROM Persons p3
                      WHERE ((p2.vte < p3.vts AND
                              p3.vts < p1.vte)
                             OR
                             (p2.vte < p3.vte AND
                              p3.vte < p1.vte)))
    UNION
    /* p1 includes p2: First */
    SELECT p1.vts AS vts, p2.vts AS vte
    FROM Persons p1, Persons p2
    WHERE p1.vts < p2.vts AND
          p2.vte < p1.vte AND
          NOT EXISTS (SELECT *
                      FROM Persons p3
                      WHERE ((p1.vts < p3.vts AND
                              p3.vts < p2.vts)
                             OR
                             (p1.vts < p3.vte AND
                              p3.vte < p2.vts)))
    UNION
    /* p1 includes p2: Second */
    SELECT p2.vts AS vts, p2.vte AS vte
    FROM Persons p1, Persons p2
    WHERE p1.vts < p2.vts AND
          p2.vte < p1.vte AND
          NOT EXISTS (SELECT *
                      FROM Persons p3
                      WHERE ((p2.vts < p3.vts AND
                              p3.vts < p2.vte)
                             OR
                             (p2.vts < p3.vte AND
                              p3.vte < p2.vte)))
    UNION
    /* p1 includes p2: Third */
    SELECT p2.vte AS vts, p1.vte AS vte
    FROM Persons p1, Persons p2
    WHERE p1.vts < p2.vts AND
          p2.vte < p1.vte AND
          NOT EXISTS (SELECT *
                      FROM Persons p3
                      WHERE ((p2.vte < p3.vts AND
                              p3.vts < p1.vte)
                             OR
                             (p2.vte < p3.vte AND
                              p3.vte < p1.vte)))) agg_table
  WHERE Persons.vts (+) < agg_table.vte AND
        agg_table.vts < Persons.vte (+)
  GROUP BY agg_table.vts, agg_table.vte;
```

Code Listing 11 - *Aggregation Written in Augmented SQL*

```
SELECT COUNT(Persons.SSN), agg_table.vt AS vt
FROM Persons p,
     TABLE(CAST(ConstantRegion('Persons', 'vt') AS ag_tab)) agg_table
WHERE p.vt.Overlaps(agg_table.vt) = 1
GROUP BY agg_table.vt;
```

Query	**SQL92**	**Augmented SQL**	**Pct. Saved**
Join	41	6	85%
Set-difference	45	37	18%
Coalescing	23	21	9%
Time-slice	4	3	25%
Aggregation	108	5	95%
	221	72	67%

Not taking into account the distribution of use among the different query types, we can see from the table that temporal augmented queries on average is one third the size of temporal SQL92 queries. It is especially join and aggregation queries that is optimized by the augmentation, but all queries benefit.

This concludes the discussion of reducing query complexity using user-defined data types available in a cartridge. The matter of specifying such data types and index support for them is the topic of the next chapter.

Chapter 3

Cartridge Design

In this chapter, we describe the actual implementation of the temporal cartridge along with the Oracle concepts used in the implementation. First we describe the object hierarchy, then the index types, and finally how it is possible to interface with the query optimizer.

As already mentioned the chosen platform is Oracle's ORDBMS. An other major ORDBMS could have been chosen for the task. The three major databases, Oracle8i, Informix Universal Server [DLM97], DB2 Universal Database [Dav00] all have an extensibility framework available which enables the specification of user defined data types, indexes and cost-based optimization.

To experiment with the Oracle extension interface, we have implemented the following. The four simple data types, *instant*, *interval*, *relative instant*, and *period* are implemented as UDOTs. The three indexes, *map21*, *map21-2*, and *hilbert* are implemented. The *map21* index is based on a simple space-filling curve technique, the *map21-2* extends this approach by partitioning the indexed periods, and finally the *hilbert* index is based on the hilbert space-filling curve.

In total, and disregarding comments, the cartridge consists of approximately 5.200 lines of PL/SQL code.

3.1 User-Defined Object Types

This section describes how the object-relational extensibility interfaces are used to declare new object types that serve as a basis for the temporal cartridge.

3.1.1 Extensible Type System

Traditionally database applications have been concerned with accessing data which is stored in tables using conventional data types such as `INTEGER`, `DATE`, or `CHAR`. Today the trend is moving towards exploiting object-relational properties of ORDBMSs by moving data into user defined object types (UDOT). Oracle supports such UDOTs along with numerous other data types, such as collections (VARRAYS and nested tables), relationships (REF), large objects (BLOB and CLOB), and external files (BFILE) [RRM99].

UDOTs are used to extend the modeling facilities of the database and to impose structure on the data stored in it. UDOTs are analogous to the concept of classes in the world of object orientation.

In the following we examine the possibilities for specifying UDOTs in the Oracle ORDBMS[1]. User-

[1]All comments regarding the status of and limitation in the Oracle DBMS is related to Oracle 8.1.6

defined object types consists of one or more attributes and optionally also a number of member and map methods. Attributes may be any of Oracles data types including other UDOTs. Member methods are procedures or functions that can manipulate the data contents of the object. Map methods are used to compare and order objects of the given type. The methods on an object can be implemented in PL/SQL or be linked to stored Java methods or external C functions.

SQL constructs are available in the DBMS extensibility interfaces for declaring, modifying and otherwise managing objects and object types. It is possible to store objects in the columns of a table and to use objects as parameters for functions and procedures.

3.1.2 Limitations of the DBMS Extensible Type System

The object-relational technology in the Oracle ORDBMS has a number of limitations, some of which concern the temporal cartridge designed. Specifically the following points have made an influence on how we designed and implemented the cartridge.

- For each UDOT a constructor is implicitly available. The constructor is named after the object and takes as parameters the same types as the attributes listed in the object specification. No other constructors can be defined ([FP97], page 616). This has forced us to create a set of stand-alone object constructor functions and violates the object oriented design of the object framework.
- Access to attributes and member methods cannot be restricted, which violates object-oriented principles of data encapsulation ([FP97], page 597). In spite of the fact that missing data encapsulation is not crucial, it might encourage future users to bypass the available methods and in stead rely on internal specifics.
- It is not possible to use object-oriented constructs such as inheritance and polymorphism ([RCG$^+$99] page 18-33). This means that an unnecessary large number of methods need to be specified.
- In PL/SQL it is possible to immediately use an object returned from a function, this is not possible in SQL. This prevents constructs like the following "t1.vt.Move('10 days').Overlaps(t2.vt)", where *Move* returns a *Period*, whose vt has been moved 10 days [LO99].
- Objects cannot be used as keys ([LO99] page 7-359). This limits our index design as we have to revert to using dates in stead of *relative instants* (explained shortly) in our open ended tables (see Section 3.2).
- PL/SQL variables of user-defined types cannot be bound into dynamic SQL statements as native data types can ([FP97] page 949). This impacts the implementation of our indexes, as the dynamically generated internal queries contain periods that must be unfolded into its conventional data type constituents.

3.1.3 The New Object Types

Seven temporal object types are specified for the cartridge. Schematically the object types are ordered in a hierarchy as shown in Figure 3.1. The basic types, *instant* and *interval* are placed in the top of the figure and is the basis of all other object types. A *relative instant* is a specialization (shown by the arrow in Figure 3.1) of an *instant*. In spite of the fact that the Oracle DBMS does not support such specializations, the concept of a *relative instant* being a specialization of an *instant* is still conceptually true. The *relative instant* is also associated (shown by a line in Figure 3.1) with an *interval* and an *instant container*. *Periods* are associated with exactly two *relative instants*. *Instant containers*, *interval containers*, and *period containers* may hold an arbitrary number of *relative instants*, *intervals*, and *periods* respectively.

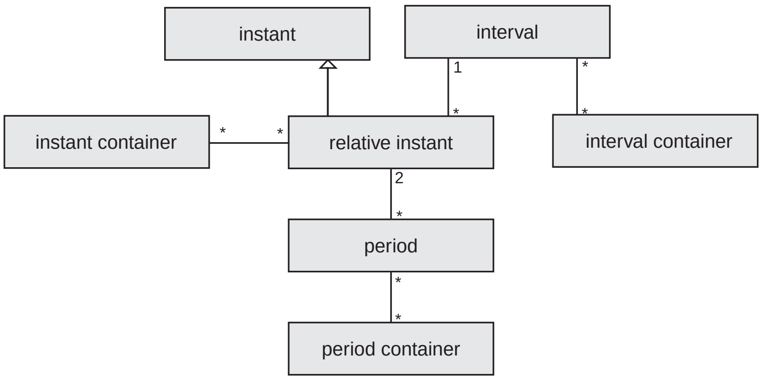

Figure 3.1: *Hierarchy of Temporal Object Types*

The following list contains a description of the seven object types. The list shows what data they contain, gives examples of them, and points out were they may be used.

Instant Instants are used to model anchored points in time. Examples of instants are "March 2000" or "2000-03-27 12:30:00" at granularities month and second, respectively. An instant is implemented as a positive number which represents an amount of granules that have passed since some predefined point in time, and a granularity specifying the type of granules. Instants may for example be used to store temporal information about meteorological samplings of temperature or humidity at different points in time.

Interval Intervals are used to represent durations of time, that are not anchored to the time line. Like *instants*, they are implemented as a count of granules and a granularity. Examples include "1 month" and "3 picoseconds". Intervals may be used to store how long a patients penicillin treatment was.

Relative instant Relative instants are much like instants only that they can be specified relative to some anchored point in time or take on special values like *now*. They are implemented as a type, an optional *instant* object and an optional *interval* serving as an off-set. The type determines if the specific instant is a special value, or a conventional relative instant. Examples are thus "March 2000 - 1 month" which represents February, year 2000, or "*now*" which represents the special temporal value. Relative instants are used in the same way as instants.

Period A period is a duration of time which is anchored to the time line. *Periods* are implemented as two *relative instants*. Examples of *periods* are "June 2000 - August 2000" which would be a period containing the three summer months of the year 2000. Examples of the use of periods include when an employee was working for a company, or when an apartment was vacant.

Instant Container An instant container is a multi set of (relative) instants and can for example be used to register all days that an employee was absent from work.

Interval Container Interval containers are multi sets of intervals and may be used to store information about which valid contract durations exists in a contracting organization.

Period Container Period containers are multi sets of periods and can be used to store information about when a given fact was true.

Each data type contains a number of attributes and methods. The underlying semantic details of these attributes and methods are specified in greater detail in Appendix A.

3.2 Indexes

This section describes the index part of the temporal cartridge. First we describe the domain index interface of the Oracle cartridge technology [RRM99]. Then we describe two index types, both from a theoretical and from an implementation point of view.

3.2.1 Extensible Indexing

Through the cartridge technology, Oracle provides an interface for creating custom index types for UDOTs. This interface was spawned by a growing need to store more advanced data types, types that could not readily be indexed with standard tree structures ([RRM99] page 7-3).

Addition of a domain index to Oracle is done by creating a new UDOT which has a predefined set of methods. When this UDOT is created, a `CREATE INDEX TYPE` statement is used to register the index and which operators it can handle. The methods that Oracle uses to control the index can be divided into four sections.

- **Definition:** The definition methods are used to create, alter, truncate, and drop an index instant. These methods have no transaction restrictions, and as such are free to use DML and DDL statements.
- **Maintenance:** These methods are used to maintain the content of an index instant, and include methods for inserting, updating, and deleting content from an index instant. These methods are only allowed to use DML statements and are not allowed to read or modify the base table on which the index is created.
- **Scan:** The scan methods are used to evaluate predicates using an index instant. Given a predicate with arguments, these methods return the ROWIDs of all rows where the predicate holds true. These methods are only allowed to execute DML query statements.
- **Meta data:** The meta data methods are used by the Oracle export utility to retrieve information about the index, that can later be used to restore the index.

For a thorough description of definition, maintenance and meta data methods see [RRM99]. Scan methods are essential elements in the fact that they serve as index-based implementations for evaluating predicates with operators. A description of the scan methods follows.

3.2.2 Index Scans

When an index scan is initiated by a user query, the first method called is *ODCIIndexStart*. The arguments to this method is among others the predicate and arguments from the user query, and the name of the index being used. The *ODCIIndexStart* method initiates the index scan, and readies the index for incrementally fetching the result.

After *ODCIIndexStart* has finished, *ODCIIndexFetch* is called. This method incrementally returns parts of the result to the query engine of the DBMS. The result is the ROWIDs that match the predicate of the user query. The state of indexes is transfered between *ODCIIndexStart* and subsequent calls to *ODCIIndexFetch* through an index type object. This means that it is the caller (DBMS query engine) that has the responsibility of maintaining the index state rather than the index itself.

When the entire result set has been returned to the DBMS, *ODCIIndexClose* is called. This method cleans up after *ODCIIndexStart* and *ODCIIndexFetch*.

3.2.3 MAP21

The MAP21 index [ND98] is an index based on a space-filling curve, and is used to index periods. The idea behind indexes based on space-filling curves is to transform the two-dimensional points they index into one-dimensional values that can be indexed by conventional indexes.

Transformation Function

The transformation function transforms a *Period* into a scalar value that can be indexed by a conventional index. The MAP21 transformation function is as follows.

$$T = ls(S, \gamma) + E$$

Here T is the scalar value, and S and E are the start and end time-points of the *Period* respectively. γ is the maximum number of digits used to represent a time-point, and ls is a function that shift it's argument γ digits to the left. The number of digits needed to store T is $2 \times \gamma$. If we transform the *Period* [2000-01-01, 2001-01-01] we get the following result.

$$ls('20000101', 8) + 20010101 =$$
$$2000010100000000 + 20010101 =$$
$$2000010120010101$$

As shown in Figure 3.2, this transformation function results in a mapping where the two-dimensional locality is poorly preserved. Even when two points are close in the one-dimensional mapping, there can be a large distance between the two corresponding points in the two-dimensional space. The reason for this is the large jumps in the MAP21 path, as seen with the jump from cell 5 to cell 6 in Figure 3.2

Figure 3.2 shows a 5x5 two-dimensional space which is mapped to one dimension by a MAP21 space-filling curve. Each cell in the two-dimensional space is assigned a unique index determined by the path, such that the first index is placed in the origin of the path.

The transformation of the special temporal value *now* is not handled by the general function, but is treated specially and will be explained shortly.

Query Translation

Because of the information stored in the index, it is necessary to translate the predicates from *Periods* to a range of MAP21 values before querying the index. The mapping from *Periods* to a range of MAP21 values that need to be fetched from index is dependent of the predicate being evaluated. The following is a description of how it is done for the overlaps predicate, similar mappings can be made for precedes, succeeds, contains, and includes [ND98].

Code Listing 12 - *Use of Overlaps Operator*

```
SELECT *
FROM t1
WHERE t1.vt.Overlaps([2000-01-01, 2001-01-01]);
```

Given a query as shown in Code Listing 12, it is necessary to translate the *Period* to a range of MAP21 values before we can query the index. The translation has to take into account, that it should encompass all MAP21 values that could possible overlap with [2000-01-01, 2001-01-01]. If nothing

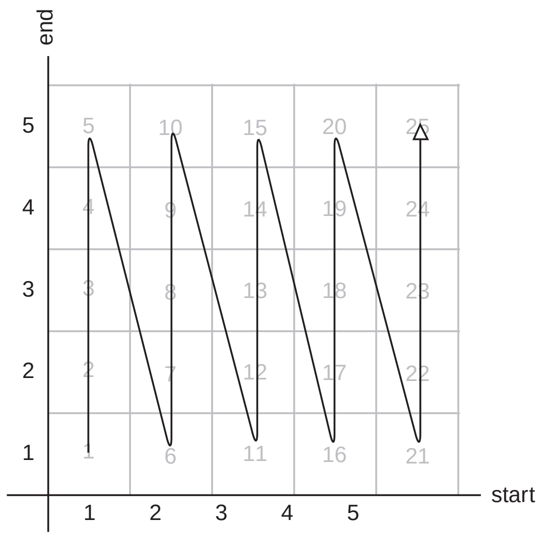

Figure 3.2: *The Mapping Order of MAP21 Transformation Function*

was known about the length of the periods being indexed, all periods starting before 2001-01-01 could possible overlap. This case can be avoided by keeping track of the longest period being indexed (which we will call Δ), with this knowledge the range of MAP21 values that could possible overlap is shown as a grey rectangle in Figure 3.3.

Only the area above the diagonal in Figure 3.3 is interesting, because points below the diagonal represents invalid periods (that end before they begin).

The periods that could possible overlap, are those starting between 2000-01-01 - Δ to 2001-01-01, and ending between 2000-01-01 to 2001-01-01 + Δ. Translated into MAP21 values, we need to examine the values between [2000-01-01 - Δ, 2000-01-01] to [2001-01-01, 2001-01-01 + Δ].

Because the previously mentioned region covers all the periods that could possible overlap, it is necessary to check periods that either starts before S or ends after E for actual overlap.

Implementation

The implementation of the MAP21 index type uses one table for meta data and two other tables per index instant, the meta data stored for each index is as follows.

tmpidx_MAP21	
Name	**Description**
Name	The name of the index, including schema
Dest_table	The name of the base table on which the index is created
Max_length	The length of the longest period being indexed

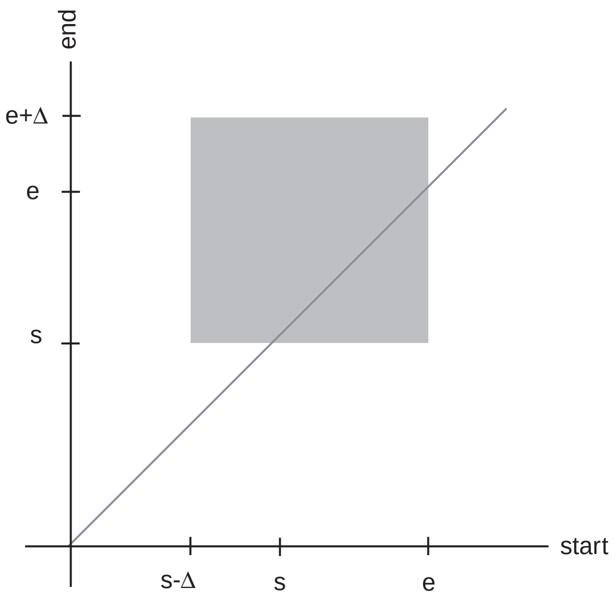

Figure 3.3: *Possible Overlapping Periods*

For each index we create two tables, one for now-relative periods, and one for non-now-relative periods. The table containing the now-relative periods is an index-organized table [RCG+99] with the following schema.

indexname_oet	
Name	**Description**
Start	The start of the now-relative period
Seq	A unique sequenced number
r	The ROWID of the tuple containing the period

Because we know that all periods in the now-relative table ends *now*, it is only necessary to store the start point of the periods. The start and sequenced number is used to define a composite primary key for the now-relative table.

The non-now-relative periods are also stored in an index-organized table, this table has the following schema.

indexname_pid	
Name	**Description**
MAP21	The MAP21 value for the period
Seq	A unique sequenced number
r	The ROWID of the tuple containing the period

The MAP21 value and the sequenced number is used to define a composite primary key.

When an index scan is initiated, the *ODCIIndexStart* method opens two cursors. The SQL for these two cursors are shown in Code Listings 13 and 14.

In Code Listing 13, **X** and **Y** refers to the MAP21 values of the lower left corner and upper right corner of the search area respectively, as shown in Figure 3.3.

Code Listing 13 - *SQL Code For Querying Non-now-relative Table*

```
SELECT r
FROM indexname_pid
WHERE map21 >= X AND
        map21 <= Y AND
        MAP21Overlaps([search period], map21) = 1
```

Code Listing 14 - *SQL Code For Querying Now-relative Table*

```
SELECT r
FROM indexname_oet
WHERE [search period].start <= SYSDATE AND
        start <= [search period].end
```

The MAP21Overlaps function takes a period and a map21 value and checks for overlap. The function is used to eliminate false hits.

As shown in the previous section, the length of the longest period in the index has a large impact on the performance of the index. This has lead [ND98] to propose an alternative structure of the index. The idea is to split the periods being indexed into three distinct tables, one which contains all the short periods, one which contains all the long periods, and one which contains the *now*-relative data. This setup prevents the case where one long period impacts the search performance of the whole index. In addition to the original implementation, where no partitioning on period durations is performed, we have implemented this alternative setup. This has been done by adding another non-now-relative table of the same structure as the present one (indexname_pid), and by adding two extra columns to the meta data table, namely the maximum length of the new non-now-relative table and the length at which periods are considered long.

3.2.4 Hilbert Index

The Hilbert index is, like MAP21, based on a space filling curve, and the structure of the Hilbert index type is also much like that of the MAP21 index type.

Transformation Function

The Hilbert transformation function has several properties that make it suitable for use in an index. First it is an optimal space-filling curve, in the fact that it has the optimal preservation of locality in the mapping between two dimensions and one dimension [LKC99]. Secondly, it has a tree structure, making it possible to adjust the complexity of the query at the expense of accuracy.

The Hilbert index type logically divides the indexed domain into a quad-tree structure [Sam84]. The idea is as follows, the domain is divided into four parts, each of these parts again divided into four parts. This division is continued until each of the parts has an appropriate size, the results of this division is a grid of cells covering the entire space. The Hilbert function is then used to define an order for these cells, this order is shown in Figure 3.4.

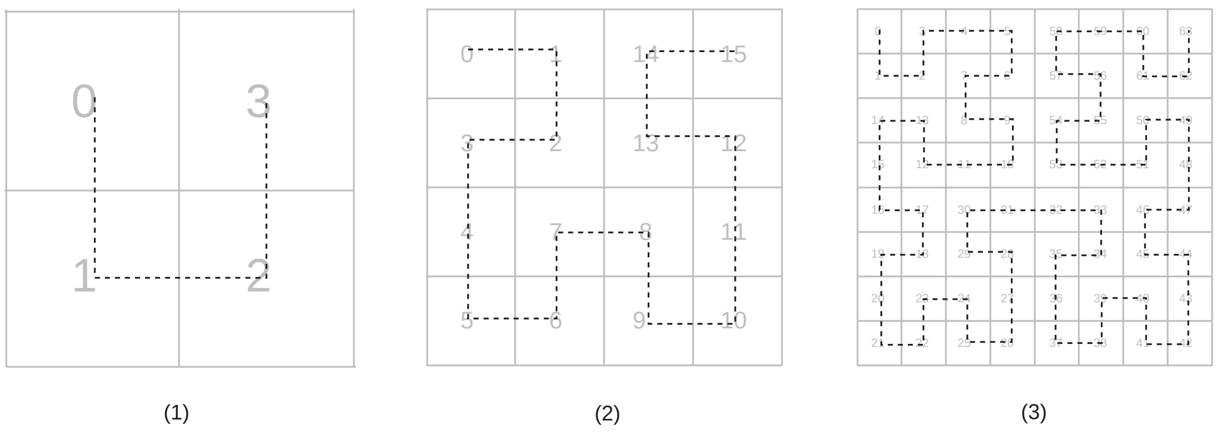

Figure 3.4: *Hilbert Curves of levels 1 (1), 2 (2) and 3 (3)*

For a description of the algorithm used to calculate the placement of a point within the Hilbert order, see [Gut99]. The algorithm starts at the top level of the logic quad-tree structure, and progresses down the tree. At each level the algorithm calculates which cell contains the search point, and progresses down that path.

Query Translation

Because of the quad-tree structure of the Hilbert index, it is possible to adjust the precision of the index query and thereby reduce the complexity of the index scan.

As with the MAP21 index, each query can be translated into a query region, an example of such a query region is shown in Figure 3.5. As the figure shows, the Hilbert ordering can enter and exit the query region several times, each of these visits results in a range of Hilbert values that is contained in the query region. When all ranges are found, they are used to search the index-organized table for periods in the query region.

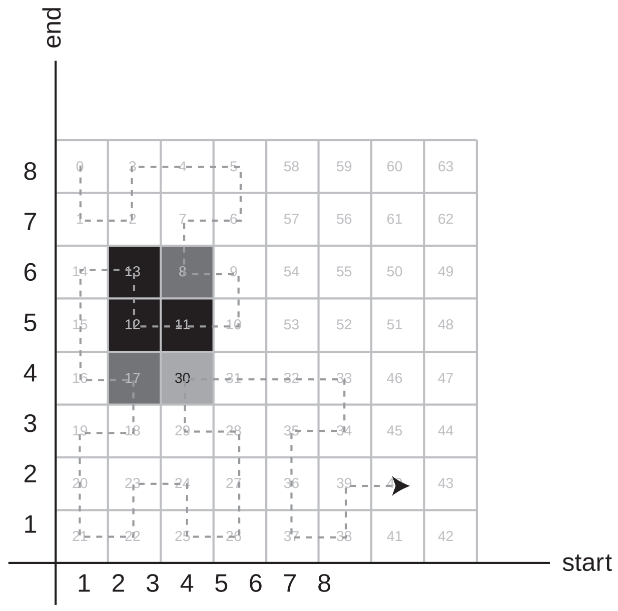

Figure 3.5: *Example of a Query Region over a Level 3 Hilbert Space*

The number of ranges in a query region is dependent on the size of the region and the size of the cells. If the region gets too big, or the cells too small, the number of ranges become substantial. It is possible to avoid this, by using the quad-tree structure of the Hilbert ordering. As each step of the recursive algorithm is used to refine the result, it is possible to stop before the algorithm hits the bottom of the logical quad-tree structure. Thereby getting a result with fewer ranges, but which includes more Hilbert values than necessary. These superfluous values are eliminated by an extra predicate (as shown in Code Listing 15 on the facing page).

Implementation

Because the overall structure of the MAP21 index type and the Hilbert index type is so much alike, their implementations are also much alike.

Like MAP21, the Hilbert index type stores two types of data. One is the meta data associated with an index, and the other is the actual index data. The meta data stored for a Hilbert index instant is as follows.

tmpidx_Hilbert	
Name	**Description**
Name	The name of the index, including schema
Dest_table	The name of the base table on which the index is created
Max_length	The length of the longest period being indexed, used for determining search areas.
Min	The lower point of the domain being indexed
Max	The upper point of the domain being indexed
Sdepth	The search depth
Tdepth	The height of the logical quad-tree

The index data stored is similar to the MAP21 index data (indexname_oet, indexname_pid), except that it is now Hilbert values that is stored instead of Map21 values. Now-relative *Periods* are also handled similarly to the MAP21 index, and will therefore not be described here.

When an index is created, an upper and lower bound is defined for the indexed domain. These bounds are time points that define the area of time covered by the index. From these bounds, the height of the logical quad-tree is calculated. The computational complexity of the Hilbert function is dependent of the height of the tree, so it is advisable not to choose a larger index domain than necessary. A search depth is also defined for the index, this depth is used when calculated overlapping ranges and defines at which level the search for ranges should be stopped.

The computation of Hilbert values, is defined as one function. The arguments to this function is a query region, the depth of the tree, the search depth, and the maximum number of ranges that may be returned. The function returns the list of ranges that are contained in the query region.

When calculating a Hilbert value for a *Period* being inserted into the index, the query region argument is a point, and the function returns a single range containing only one value, which is the Hilbert value of the specific point.

When performing an index scan, the ranges returned from the Hilbert function is used to compose a dynamic SQL statement to query the index-organized table. An example of such an SQL statement can be seen in Code Listing 15.

Code Listing 15 - *SQL Code For Querying Hilbert Index*

```
SELECT r
FROM indexname_pid
WHERE (( hilbert >= X AND hilbert <= Y) OR
       ( hilbert >= G AND hilbert <= H)) AND
       HILBOverlaps([search period], hilbert) = 1
```

The query in Code Listing 15 have two ranges, one from X to Y and another from G to H. *hilbert* is the hilbert value stored in the meta data tables. The *HILBOverlaps* method takes as input a period and the Hilbert value and checks for overlap in order to eliminate false hits.

3.3 Optimization

This section describes the third and last part of the extensibility interfaces used in the development of the temporal Cartridge, namely extensible query optimization.

First Oracle's extensible query optimization interface is described followed by a brief discussion of how this feature may be used in a temporal cartridge.

Extensible optimization for the cartridge has only been examined briefly[2], and has not been included in the actual implementation. The focus of this section is therefore on the extensibility interface and not a concrete implementation. The description is based on Oracle documentation including [RRM99].

3.3.1 Extensible Optimizer

The query optimizer is the part of a DBMS which has the responsibility of choosing the most efficient way of executing a query statement. Execution, for example, depends on the order in which tables and indexes are accessed. An optimizer can either use cost-based optimization or rule-based optimization.

A cost-based optimizer considers between different access paths by using statistics, e.g., in the form of histograms, about the involved database objects. The Oracle DBMS supports this kind of optimization through SQL statements such as `ANALYZE` and `COMPUTE STATISTICS`. A rule-based optimizer on the other hand chooses between access paths by considering the ranks of these access paths.

[2]A stand-alone prototype extensible optimizer was implemented for periods.

Oracle supports both cost-based and rule-based optimization. A number of features can however only be used by the cost-based optimization strategy, including extensible optimization.

The extensible optimizer allows three kinds of functions to be defined for user-defined functions and indexes: *statistics collection* functions, *selectivity* functions, and *cost* functions.

All extending of the optimizer is done by declaring functions that the optimizer calls when appropriate. Such functions are specified in an object implementing the *ODCIStats* interface. This object is registered with the query execution engine using the `ASSOCIATE STATISTICS WITH` command. Each of the three functionalities in the extensible optimizer is explained below.

Statistics Collection Functions

Statistics on database columns and indexes are collected using the `ANALYZE` command. With the introduction of user-defined domain indexes the DBMS cannot, on its own, collect statistics on such indexes, because it does not know the internal structure of the index.

In the light of this problem the optimizer has been extended to let users define and associate custom statistics collection functions (SCF). SCFs can be associated with individual columns, object types, index types, and domain indexes. The SCFs are called by the optimizer whenever a domain index or column is analyzed. The statistics generated by the user-defined SCF is anonymous to the DBMS, in the fact that it has no knowledge of its structure, representation, or meaning. Any interpretation of the statistics is done in the user-defined query optimization functions. In the case of table columns and object types SCFs are called whenever an appropriate column is analyzed. If the data type of the column is native to the DBMS, the statistics generated by the SCF is collected along with the conventional statistics. Two functions must be specified in connection with the statistics gathering part of the extensible optimizer object. The first, *ODCIStatsCollect*, collects the statistics when the `ANALYZE` command is issued. The other, *ODCIStatsDelete*, deletes the statistics when the `ANALYZE DELETE` command is issued. Both *ODCIStatsCollect* and *ODCIStatsDelete* are overloaded in order to work with both table columns/object types and with user-defined domain indexes.

Selectivity Functions

The statistics gathered by the SCFs above are used to determine the selectivity of a given query predicate. The selectivity is a measure for how many percent of the rows that are chosen by the predicate. This selectivity is in turn used to estimate the cost of a particular access method.

With extensible optimization it is possible to define custom selectivity functions (SF), which can be associated with user-defined operators, stand-alone functions, functions in packages and methods in object types. The SF is called by the optimizer each time it encounters a predicate with a user-defined operator, function, package function or object method. If we, for example, have the object method *overlaps*, associated with a SF, this SF will we called when a query contains predicates such as "`overlaps(...)  = 1`". The entire predicate is passed to the SF as an argument.

Only a single function, *ODCIStatsSelectivity*, needs to be declared in the optimization object in order to make use of custom selectivity measures with the extensible optimizer.

Cost Functions

As in the case of statistics and selectivity functions the optimizer has no way of determining the cost of a particular user-defined domain index based access method. The reason again being that the optimizer has no knowledge of the internal structure of the domain index.

Therefore it is an option to specify user-defined cost functions (CF) and associate them with user-defined stand-alone functions, package functions and object type methods. When the optimizer encounters a predicate involving a stand-alone function, package function or object type method with which a CF has been associated it initiates a call to this CF. The same is possible with domain indexes and index types except now the predicate references an operator, that can be evaluated using such a domain index.

A single function, *ODCIStatsCost*, is necessary to add custom cost calculations. This function takes as parameters a description of the operator and the arguments to this operator. The function returns a cost, consisting of two components, namely the CPU and I/O costs. The *ODCIStatsCost* function is overridden in order to support both domain indexes and function operators.

3.3.2 Use of Extensible Indexing

Turning our attention to the particular problem of developing the temporal cartridge, we have to specify functions for the three tasks of collecting statistics, estimating selectivity, and calculating cost. Statistics collection and selectivity estimation are closely linked in the fact that the statistics are used in the selectivity estimation process. Statistics and selectivity is likewise used to calculate cost.

To our knowledge no-one has treated the topic of estimating statistics for temporal data directly, but concepts of use can be found in the spatial temporal research [LKC99].

Many approaches have been given for determining the selectivity of queries [MCS88], including sampling, parametric techniques, and histograms, where input data is partitioned into a number of subsets called buckets. Research distinguishes between selectivity estimations for 1-dimensional data and for multi-dimensional data [LKC99]. According to [LKC99] histograms are well suited for data with dimensionality lower than three. Multi-dimensional selectivity estimation techniques include Hilbert numbering, multi-level grid files [PI97]. Neural networks have been suggested [LZ98] as a method for estimating selectivity on user-defined data types.

This concludes the discussion of the Oracle Extensibility Interface in regards to defining user-defined object types with the Extensible Type System and user-defined domain indexes in the Extensible Indexing feature and query optimization functions with the Extensible Optimizer functionality.

Chapter 4

Performance Test

In this chapter we evaluate the performance of the cartridge, this is to evaluated whether REQ3 is fulfilled. The evaluation is divided into two parts, first an evaluation of the index performance, and secondly a performance evaluation of the queries described in Chapter 2. In the index evaluation, we compare the performance of the indexes to each other, and to an implementation using native data-types indexed with a B^+ tree. The query evaluation will compare the performance of the augmented queries to the performance of queries using native data-types, both with and without indexes.

4.1 Test Setup

The tests are conducted on a Oracle instant equipped with the TerraTele schema described in Chapter 2. The software used for the evaluation is as follows.

- Oracle 8.1.6.1.0
- SQL*Plus 8.1.6.1.0
- Windows 2000 v5.00.2195

The hardware used is as follows.

- Processor: 400Mhz Pentium II
- Memory: 256MB
- Disk: 10Gb 4400rpm ATA

The configuration of the Oracle instant is not changed from default, which is 14793 disk buffers of 8192 bytes each.

Each test is conducted five times, the fastest and slowest times are removed, and the result is calculated as the average of the remaining three times.

4.2 Index Tests

We choose six tests to evaluate the performance of the implemented indexes. Two of these tests are aimed at tuning specific index parameters on Hilbert and Map21-2, and four are aimed at testing the ability of each index to handle different types of data, e.g. long or now-relative *Periods*. For comparison we have included an implementation using native data types and B^+ indexes.

In each measurement we have included four indexes, namely a Hilbert index labeled "Hilbert", a Map21 index labeled "Map21", a Map21 index with the *Periods* divided into a long and short table labeled "Map21-2" and finally a query using conventional data types and B^+ indexes labeled "Conventional".

For these tests we use a standard dataset consisting of 50000 tuples each associated with an *Period*, and uniformly distributed over a period of five years. 5 percent of the *Periods* are now-relative. 95 percent of the remaining *Periods* have a length uniformly distributed between 10 to 100 days, and 5 percent a length uniformly distributed between 100 to 1000 days. The size of the datasets are 3.9MB for the augmented and 2.0MB for the conventional.

The query used for these tests, is a simple overlaps query that returns all *Periods* that overlap a given *Period*. The *Period* used, cover 10 percent of the indexed time region.

4.2.1 Search Depth of Hilbert Index

One of the parameters in the Hilbert index is the search depth. This parameter controls how deep a search should go down the tree, and thereby how precise the initial selection of *Periods* is. Adjusting this parameter is a tradeoff between the complexity of calculating search ranges, and the complexity of eliminating false *Periods*.

We have included the Map21, Map21-2 and conventional index in this test, only to serve as reference marks. They are not affected by the search depth, and their performance are therefor constant.

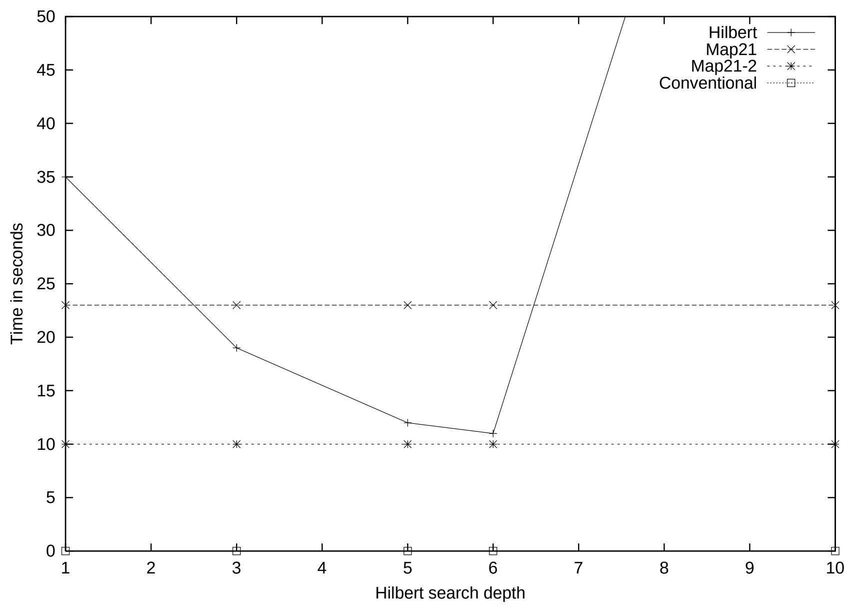

Figure 4.1: *Performance Relative to Search Depth in the Hilbert Index*

As we can see from figure 4.1 the performance of the Map21, Map21-2 and Convention indexes remain constant, while the performance of the Hilbert index improves to a certain point, at which is decreases rapidly. The increase in performance is due to fewer false tuples being included in the Hilbert ranges, while the sudden decrease is due to the time it takes to calculate the Hilbert ranges. The following table shows how long it takes to calculate Hilbert ranges at a given depth, and how many ranges are returned. The table is calculated using the same overlaps query as Figure 4.1.

Search Depth	**Ranges returned**	**Time to calculate (sec.)**
6	20	0,651
7	30	1,792
8	45	6,659
9	95	26,008
10	262	102,000

We can see from the table, that the time used to calculate the Hilbert ranges is exponential. This suggest that it is better to calculate too few ranges than too many, and that a large number of indexed *Periods* is necessary to justify going deep into the tree.

4.2.2 Index Split of Hilbert and Map21-2 Indexes

Another index tuning parameter is when to consider a *Period* long. This parameter determines the distribution of *Periods* between the short period table and the long period table in the Map21-2 and Hilbert index.

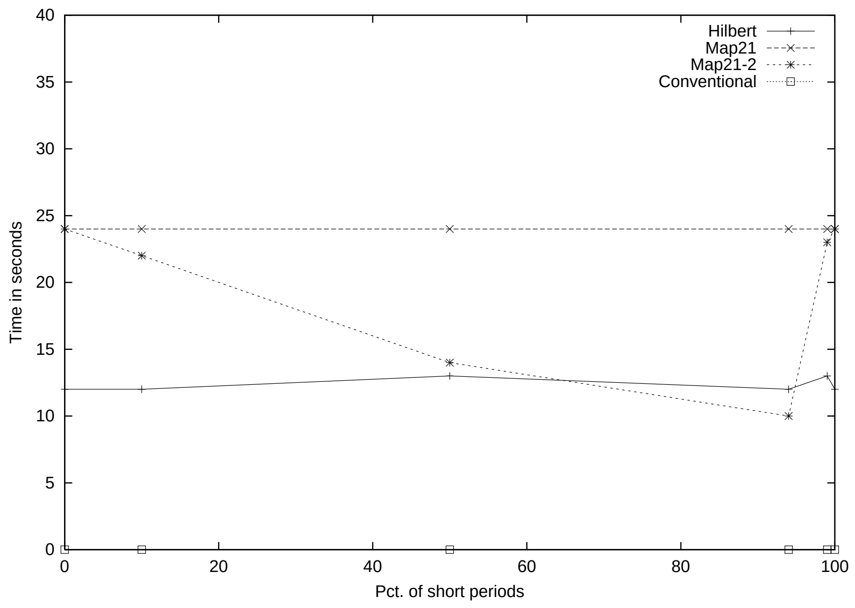

Figure 4.2: *Performance Relative to Index Split Limit*

The x-axis of Figure 4.2, is the amount of *Periods* in the data-set that is considered short, and therefore is stored in the short period table in the Map21-2 and Hilbert indexes.

As we can see from Figure 4.2, this split has a huge effect on the Map21-2, and is optimal when the small *Periods* constitute 95 percent. This is consistent with the distribution of data, where 95 percent of the *Periods* is 10 to 100 days in length, and 5 percent is between 100 and 1000 days in length. The split limit has very little influence on the performance of the Hilbert index, this may be due to the fact that most of the time spent in the Hilbert index is used to calculate Hilbert ranges.

4.2.3 Length of Periods

Long *Periods* are often a problem with temporal indexes, because they result in both an uneven distribution of *Periods* between the long *Period* and short *Period* tables, and an increasingly large search area for querying the indexes.

The data used for this test contains progressively more long *Periods*, starting with no long *Periods* and ending with only long *Periods*. Short *Periods* are between 10 and 100 days, and long *Periods* are between 100 and 1000 days.

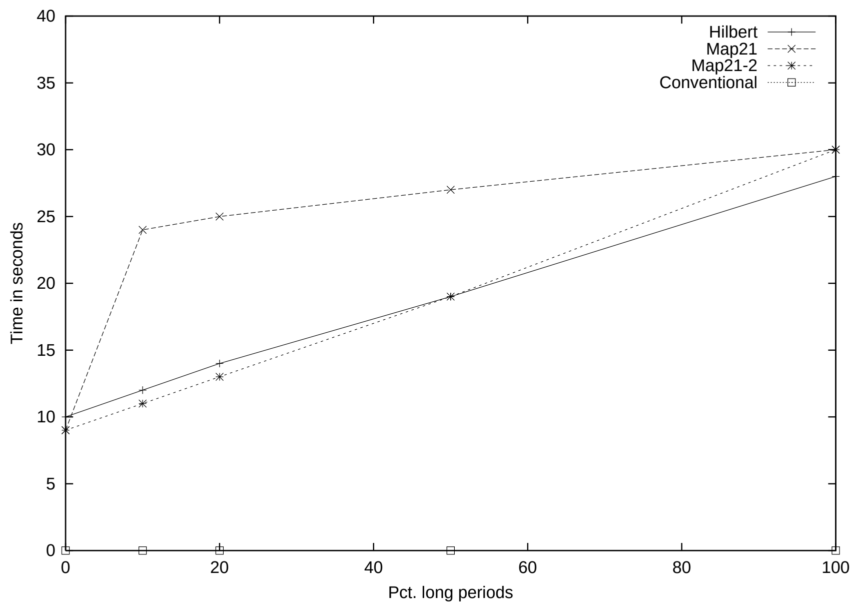

Figure 4.3: *Performance Relative to Length of Indexed Periods*

As expected, Figure 4.3 shows that the performance of Map21 degrades even with a small percentage of long tuples. This is consistent with the fact that Map21 stores all *Periods* in one table, and a single long *Period* can therefore alter the size of the search area for all *Periods*. The other indexes also suffer from a large amount of long *Periods*, but this may be remedied by changing the split limit as the amount of long *Periods* rise.

4.2.4 Now Relative Periods

The fourth test is designed to test the ability of the indexes in handling now-relative *Periods*. Each of the indexes handle now-relative *Periods* similarly, namely by keeping them separate from the non-now-relative *Periods*.

As we can see from Figure 4.4, all indexes improve as the percentage of now-relative *Periods* rise. This is because querying and indexing now-relative *Periods* is simpler than non-now-relative *Periods*. Because the end point of now-relative *Periods* is known, we only have to index the start point and no Hilbert or Map21 translation is necessary, thereby making the query simpler. As shown, all augmented indexes perform that same when indexing 100 percent now-relative *Periods*. This is expected, as all indexes handle now-relative tuples in the same manner.

4.2.5 Query Area

The fifth test is designed to test the ability of each index in handling different size query areas.

As Figure 4.5 shows, all indexes handle large query areas fairly well. Not surprisingly Map21 outperforms both Map21-2 and Hilbert when the query area approximates 100 percent of the indexed area. This is because the Map21 algorithm is simpler, but returns many false *Periods*. This is a problem

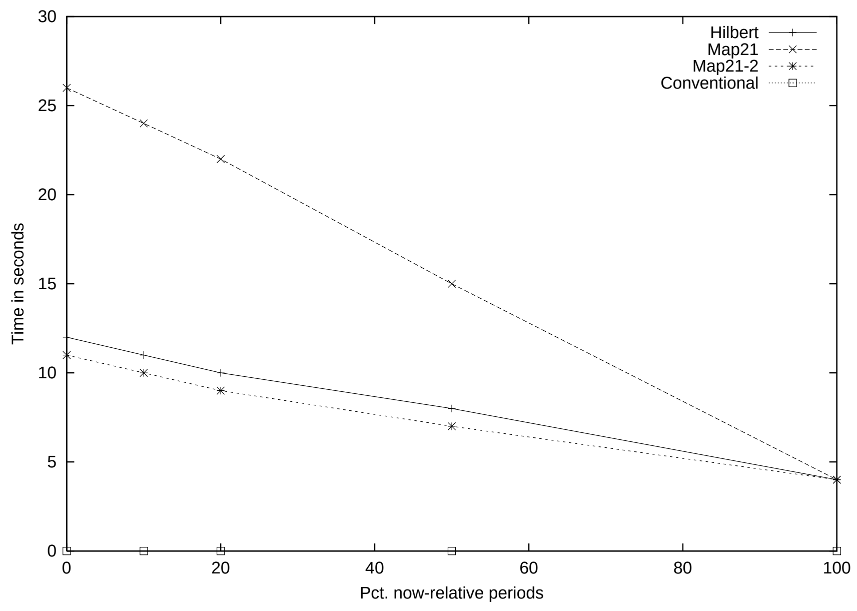

Figure 4.4: *Performance Relative to the Amount of Now-relative Periods in Index*

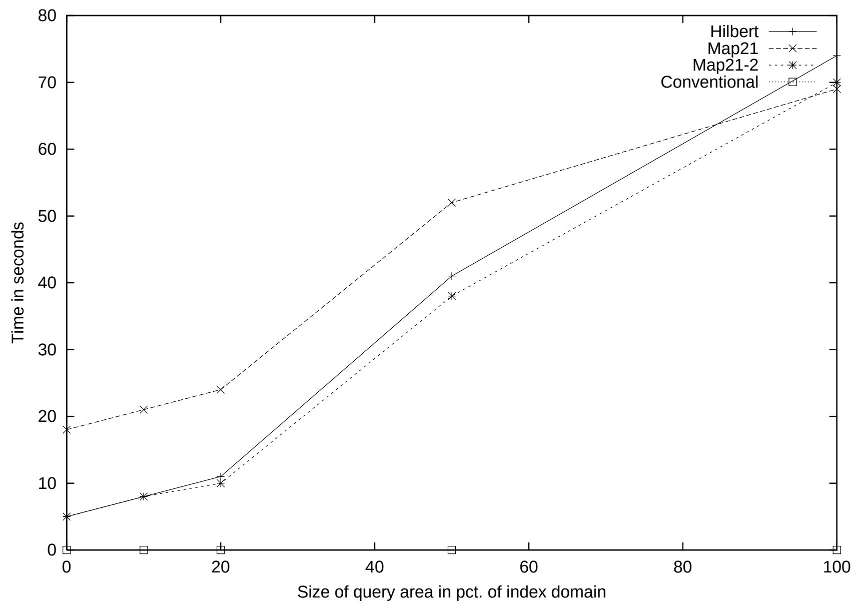

Figure 4.5: *Performance Relative to Size of Query Area*

with small queries, but with large queries the amount of false *Periods* return is small compared to the amount of true *Periods*.

4.2.6 Amount of Tuples

The last index test, test the ability of the index in handling large amounts of tuples.

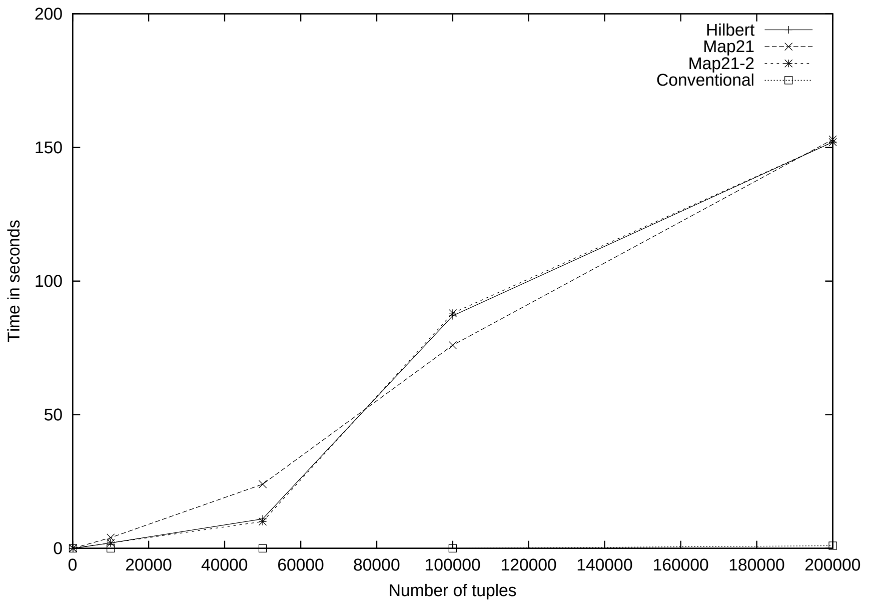

Figure 4.6: *Performance Relative to Amount of Tuples*

Figure 4.6 shows, that all indexes handle large amounts of data almost equally. Map21 is faster than both Map21-2 and Hilbert at 100000 tuples, whereas Map21-2 and Hilbert perform equally throughout the range.

4.3 Query Tests

In this section we test the performance of the queries defined in Chapter 2. Both queries using conventional data types and queries using augmented temporal data types are tested.

Each query was tested using a unique dataset without now-relative *Periods*. Now-relative *Periods* was left out of the dataset because none of the queries, based on conventional data types, support now-relative *Periods*. The algorithm used for creating the data-sets, is the same as described in section 4.2. The queries were tested using the TerraTele schema described in Chapter 2.

In the following sections each test contains six series. Four series with augmented data-types, namely "Augmented" which is without any index defined, and "Map21", "Map21-2" and "Hilbert" for the respective indexes. The last two series uses conventional data-types, and are named "Conventional" for the one using conventional data-types and no index, and "B+" for conventional data-types with B^+ indexes.

4.3.1 Join

The data-set used to test the join query contains 1000 objects in each of the tables, and is designed as described in Section 4.2.

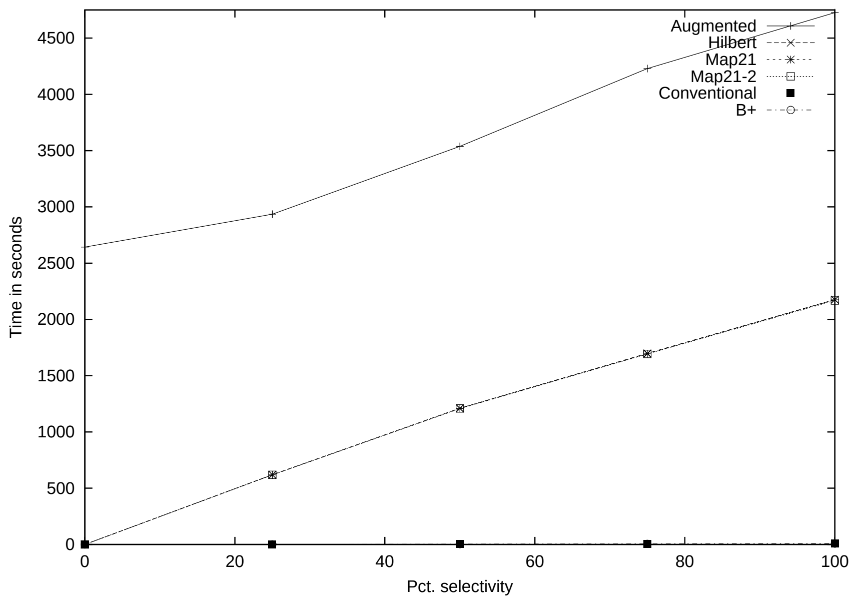

Figure 4.7: *Performance of the Join Query*

As we can see from Figure 4.7, the augmented query cannot compete with the query using conventional data types. This is because the RDBMS has no knowledge of the semantics of user defined methods (UDMs), and therefore have to check each *Period* from the *Prices* table with each *Period* from the *Calls* table, resulting in a nested loop comparison. This nested loop is not necessary with conventional data types, because nothing is hidden from the RDBMS, and it can therefor optimize the query. This optimization consists, among other things, of sorting the two tables and thereby only comparing possibly overlapping *Periods*.

By adding an index, the nested loop comparison is made a bit faster as each *Period* in the *Prices* table results in an index scan instead of a full table scan. The selectivity of the index scan is determined by the length of the *Periods* in the *Prices* table, and is in this case roughly 3 percent.

4.3.2 Set-Difference

The data-set used to test the set-difference query contains 1000 objects in each of the tables, and is designed as described in Section 4.2.

The SQL for the set-difference query using augmented data types is very similar in structure to the query using conventional data types. This similarity combined with the added overhead in working with UDTs, makes the augmented set-difference query slower than the conventional query.

4.3.3 Coalescing

The data-set used for testing the coalescing query contained 100 unique objects, each consisting of 1 to 5 tuples, for a total of 311 tuples. This resulted in a coalescing factor of 66%.

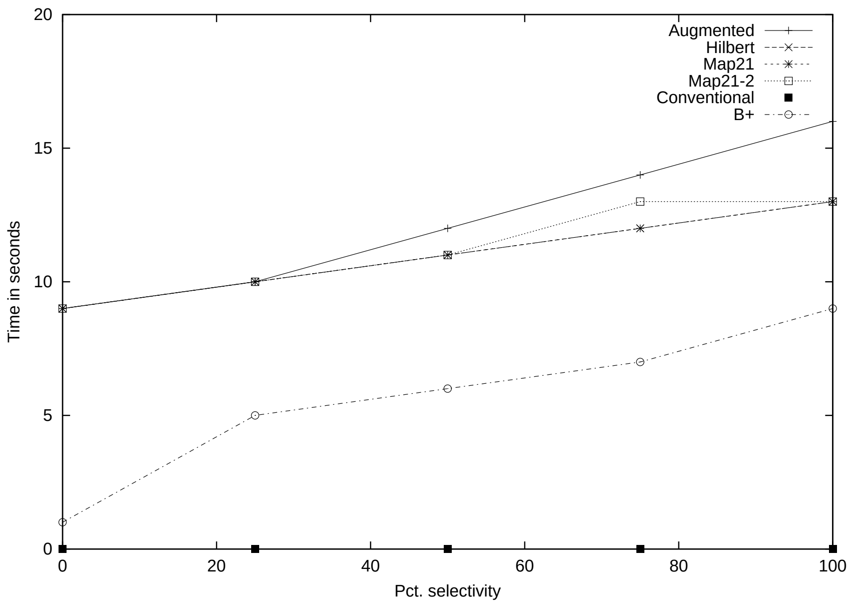

Figure 4.8: *Performance of the Set-Difference Query*

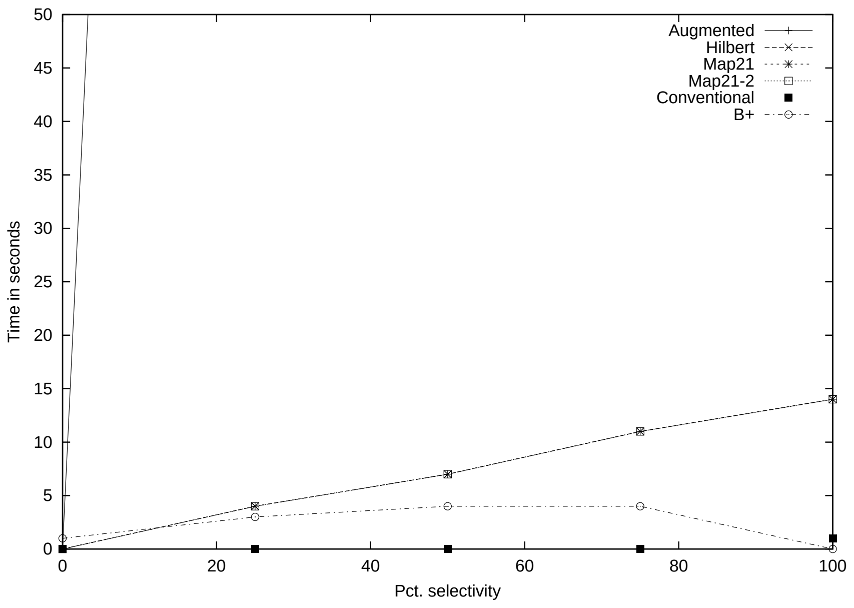

Figure 4.9: *Performance of the Coalescing Query*

As shown on Figure 4.9, the augmented index queries are several times faster than the non-indexed augmented query. Within the augmented index queries, there are virtually no difference in performance between the different indexes. This is because the low number of tuples in the table, make the performance difference of the indexes insignificant. Although the augmented index queries are faster than the non-index augmented query, they are slower than the conventional queries when the selectivity goes up.

The conventional query with the B^+ index are slower than the conventional query without the index. This could be avoided by using the cost based query optimizer, which probably would have selected not to use the index, thus making the indexed query at least as fast as the non-indexed query.

4.3.4 Aggregation

The data-set used to test the aggregation query contains 150 objects, and is designed as described in Section 4.2.

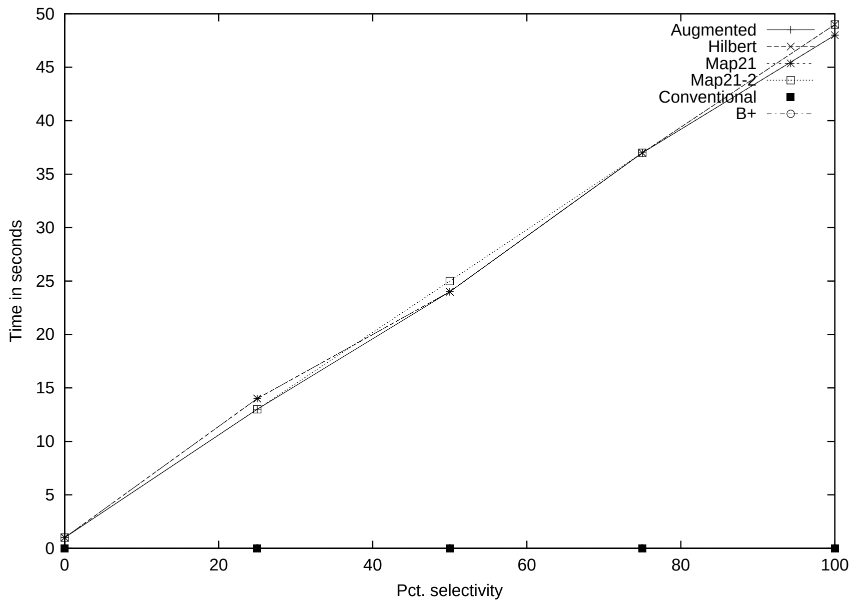

Figure 4.10: *Performance of the Aggregation Query*

The performance measurements for the aggregation query is shown on Figure 4.10, where we can see that all the augmented queries are equally fast, but the conventional queries are faster.

As expected all the augmented queries are equally fast, this is because the query does not use any index optimized operations, and therefore does not benefit from the generated index.

Even though the augmented aggregation query substantial fewer lines of code than the conventional query, it is still slower. This is because the query engine calls the *ConstantRegion* function for each tuple in the master table, resulting in a full table scan of the master table for each tuple, resulting in a nested loop.

4.3.5 Time-Slice

The dataset used to test the time-slice query contains 50000 objects, and is designed as described in Section 4.2.

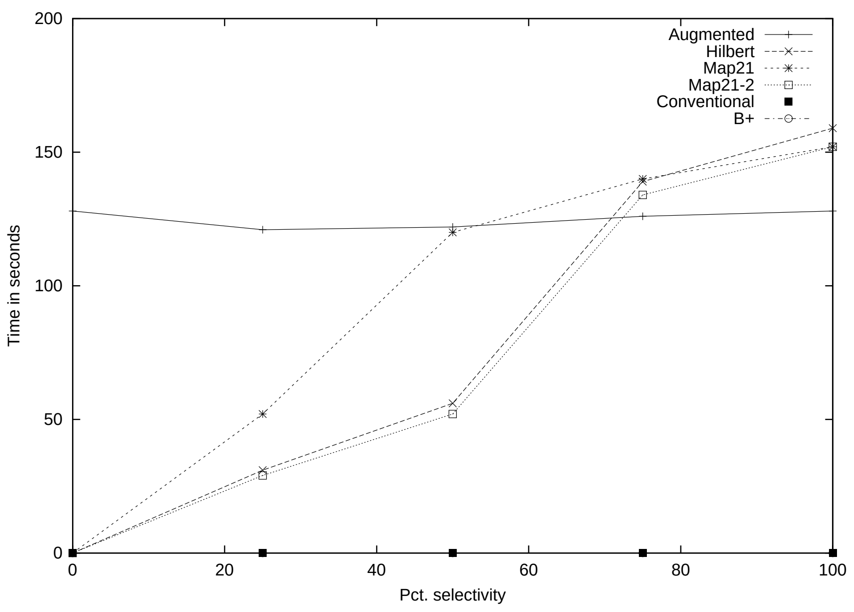

Figure 4.11: *Performance of the Time-Slice Query*

The timings for time-slice is shown in Figure 4.11, here we can see that the conventional query is consistently faster than the augmented queries. Among the augmented queries, there is little difference on Map21-2 and Hilbert, which both are a bit faster than Map21. As the selectivity rises above 70 percent, it becomes faster to use full table scans that to use indexes.

The augmented query without indexes takes the same amount of time, regardless of the selectivity. This is because regardless of the selectivity, it has to do one full table scan to retrieve the matching tuples.

4.4 Evaluation of Performance

The basis for this chapter, was to evaluate the performance of the suggested temporal frame-work. The chapter contained a test of the indexes compared to using conventional data-types and B^+ indexes, and a test of the queries suggested in Chapter 2. This performance evaluation is used, to decide whether the suggested frame-work fulfills REQ3.

The implemented temporal indexes was consistently slower than the conventional B^+ indexes, some times more than 150 times as slow. The temporal indexes did not scale as well as B^+ indexes, and the performance was more dependent on the structure of the data than the B^+ indexes.

Based on reduction in lines of code, the queries tested in this chapter can be divided into three groups. The simple queries, the ones that did not benefit from the augmented temporal data-types and finally the ones that did benefit from the augmented data-types.

The performance of the simple query, time-slice, did not benefit from the augmentation. It is a simple query, both to express and to execute, and there where little possibility for improvement by augmentation.

The queries that did not benefit from the augmented data-types was coalescing and set-difference, the structure of these queries did not change with the augmented temporal data-types. The performance of these queries where very slow compared to queries using conventional data-types, this is probably because of the added overhead of using UDTs.

The last group of queries are those, that improved structurally by adding the augmented temporal data-types, these are the join and aggregation queries. The join query, although simpler to express, was more complex to execute. The *Overlaps* operator translated into a nested loop, where the RDBMS makes an index search for each *Period* in the master table. This makes the augmented join slower than the conventional join. The aggregation query suffers from the same problem as the join query, it is executed as a nested loop. The query was expressed as a join between the virtual table of constant regions and the master table. Unfortunately the RDBMS executes the *ContinuousRegion* function for each period in the master table. This nested loop makes the augmented version slower than the conventional.

Based on the performance of the indexes and the performance of the expressed queries, we can say that the frame-work described does not meet REQ3. Queries like aggregation could improve in performance if the RDBMS checked the dependencies of the function before execution, while others like join could not easily be improved in performance.

Chapter 5

Conclusion

Concluding on the work performed and discoveries made in this project we first list the contributions of the project. Following this, we discuss how the goals were met and requirements fulfilled. The test results of Chapter 4 are summarized and related to the goals and requirements. Finally we discuss what future work is relevant with regards to our task.

The contributions of this project in relation to the field of temporal databases are as follows.

- We suggest a way of reducing the complexity of temporal queries with as much as 95% compared to the existing SQL queries. On average the suggested method saves roughly two thirds of the query measured in lines of code.
- A hierarchy of object types complying with accepted research results from over 20 years of research activities in temporal databases are designed. The object hierarchy is easily extendible to encompass concepts such as custom calendars, indeterminacy, and user defined granularities and time domains, which have so far been limited to research database prototypes.
- Theoretically described indexes are adapted to the new object types, implemented and tested as an extension to a commercial ORDBMS platform.
- The project studies Oracle's cartridge technology in relation to the implementation of valid time temporal data.

Together all contributions serve as a demonstration of a framework for adding complex temporal functionality to an existing ORDBMS by encapsulating it in the database using existing extensibility features of the ORDBMS.

GOAL1 was to examine the possibilities of easing the task of managing now-relative valid-time data in commercial ORDBMSs. It was shown that it is possible to reduce the number of lines of code necessary to express a temporal query by a factor three. This fulfills REQ2 which concerns simple code. The objective is related to REQ4 and REQ5 in the fact that the database features needed for pursuing our goal of reducing code complexity via new objects, in fact is met by major ORDBMS platforms. However these features are still not mature enough for also supporting efficient use of the temporal data types in queries.

GOAL2 was to provide a framework for efficient execution of temporal statements. It was shown that it is not possible to achieve high performance of queries based on the temporal framework in todays ORDBMS platforms. Requirement REQ1 that deals with use of only existing technology, can thus not be fulfilled along with REQ3. The performance tests indicate that the enhanced queries execute several times slower than the original counterparts.

Overall we conclude that while it is possible to reduce the complexity of temporal code, the related performance degradation is not acceptable for most applications. With little effort from the RDBMS vendors, queries like aggregation could be made faster, while others, like join, need a larger amount of work.

We think that the concept of a period might be too simple to justify being modeled as an object, and believe that more complex objects, such as temporal containers, are more suitable. The temporal containers have several benefits, among others they allow table operator like functionality, and allow for more object oriented schema design.

Future Work

In this rapport we have described some of the challenges of working with temporal data in ORDBMSs, but before it is possible for companies to use this technology a number of tasks have to be done.

We have created three indexes, all based on space-filling curves. These are just one type of temporal indexes, and it might prove useful to explore other index types, e.g., the R-Tree contained in the spatial cartridge [Coo99].

The user-defined objects described in Chapter 3 is designed and implemented to support multiple calendars. The calendar system has not been implemented, but we feel that the added complexity of dealing with user defined calendars might justify the added overhead from the temporal cartridge.

Several researchers, within both the temporal and spatial research community, have worked with indeterminate data. The user-defined objects defined here, would be able to encapsulate this indeterminacy, thus making it almost transparent to the user.

The current implementation of the temporal cartridge is done completely in PL/SQL, which is only one of the languages supported by Oracle. Because some of the methods are processor intensive we expect that they are faster when implemented in C or Java.

Appendix A

Semantics

This section introduces the semantics for the temporal framework applied in the temporal cartridge developed. The appendix has the following structure: First it introduces the general semantics of granularities, operations on basic temporal data types, and the new cartridge object types them selves. Then a more detailed semantics of the operators on the object types are given.

A.1 General Semantics

A point in time can be seen as a point on a continuous time line [DELS98], beginning at the Big Bang [Wal92] and ending at the end of the universe. However, since databases store discrete values, we adopt a discrete representation of time. This discrete representation of time is based on units called instants and are related with the matter of granularity, calendars and indeterminacy as explained in the following.

Temporal data types such as instants are associated with a granularity which specifies to what precision the information can be interpreted. A *granule* is a subset of the time domain and the *granularity* can be defined as a mapping G from the set of integers $\mathbb{N}$ to granules, such that the granules inside the granularity are non-overlapping and totally ordered [DELS98].

A relationship exists between granularities, in the fact that the granules of a given granularity can be aggregated into new granules of a coarser granularity. Similarly a granularity may be finer than an other. If a granularity is coarser or finer than an other granularity, we say that the two granularities are comparable. The finer-than and coarser-than relationships have been described as follows.

> If G and H are two comparable granularities we can say that
>
> - H is *coarser* than G ($G \trianglelefteq H$)
> - G is *finer* than H ($G \trianglerighteq H$)
>
> if for each granule $h \in H$ there exists a set of granules $S \subseteq G$ such that $h = \bigcup_{g \in S} g$

An example of these relationships include the facts that a month is finer than a year (month $\trianglelefteq$ year) and that an hour is coarser than a minute (hour $\trianglerighteq$ minute), whereas a week is neither coarser nor finer than a month. The reason for this is that a month is not composed of an integer number of weeks.

Granularities are given with respect to calendar. We leave the treatment of calendars to others ([SS94]) except for the brief introduction in this paragraph. The above mentioned granularities are, for example, those of the *Gregorian calendar.* A calendar is thus a partitioning of the underlying time line into granules [DS94]. It provides a mapping between granules and character strings and functionality for handling such granularities.

A.2 Data Types

As outlined in Section 3.1.1 seven object types are defined for the system. Schematically the data types are ordered in a hierarchy as previously shown in Figure 3.1 on page 25. Semantics for the seven object types are listed in the following paragraphs.

A.2.1 Instant

A time point is the actual moment an event occurs and is modeled in the discrete framework by an *instant.* Instants are either determinate or indeterminate. Indeterminate instants store un-precise or "do-not-know-when" information, whereas determinate instants store precisely known information.

All instants are composed of a sequence of granules, which is called the *support.* The granules in the support are the granules g in granularity G during which the time point of the given event may exist. If the lower granule of this support is equal to the upper granule (i.e. the support consists of exactly one granule) it is a determinate instant, otherwise indeterminate.

In the following we use the notation $g = l_G \sim u_G$ for an instant g in granularity G. l_G and u_H are the first and last granule of the support respectively. We thus have that

$$l_G \sim u_G = \{g \in G \mid l \leq g \leq u\}$$

In the following, instants are referred to by the symbol "ins". An indeterminate instant a and an determinate instant b is illustrated in Figure A.1 below.

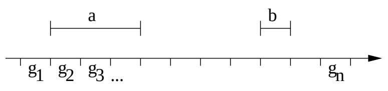

Figure A.1: *Indeterminate Instants a and b in Granularity G*

In the following instants (***Ins***) are determinate and contains the following:

- Granule index $i \in \mathbb{N}$
- Granularity $G \in \mathbb{G}_{cal}$, where $\mathbb{G}_{cal}$ is the set of granularities in calendar *cal*.

Instants (and other data types) are written with a subscript denoting their granularity, e.g. 2000_{years} or 2000-15-03_{days}. The following are examples of determinate instants.

2000-01-01_{days}	The date 1 January year 2000.
1976-10-26 09:20:$00_{seconds}$	The 26 October 1976 at 9:20am.
1900_{years}	The year 1900.

A.2.2 Interval

Intervals represent unanchored periods of time. If associated with a the valid-time of a fact, intervals contain only information about the length of time the fact was valid, but no information about when it was. An interval can be both forward and backward pointing.

Intervals are much like instants, and consists of a granule count and a granularity. We define intervals as a signed number of granules in some granularity G.

The determinate intervals ($\boldsymbol{Inv}$) are referred to by the symbol "inv" in our semantics and consists of:

- Granule count $i \in \mathbb{N}$
- Granularity $G \in \mathbb{G}_{cal}$, where $\mathbb{G}_{cal}$ is the granularities in calendar cal.

The following are examples of determinate intervals.

1_{weeks}	One week.
7_{days}	7 days forward from some instant.
-8_{hours}	8 hours backward from some instant.

A.2.3 Relative Instant

A *relative instants* of granularity G consists of an interval and an instant both of granularity G.

A relative instant may be of a special type, e.g., *now*, which means that the instant is not bound to a time value until it is used.

Relative instants ($\boldsymbol{Ri}$) contain:

- **ins**: Instant $(i, G) \in \boldsymbol{Ins}$
- **inv**: Interval $(i, G) \in \boldsymbol{Inv}$

The following are examples of relative instants.

now_{days} - 1_{days}	Yesterday.
now_{days} + 01-00$_{days}$	Same day next month.
2000-01-01$_{days}$ - 7_{days}	1999-12-25$_{days}$.

A.2.4 Period

We define a *period* per of granularity G to be a contiguous subset of the time domain between two instants i_1 and i_2 represented by granules g_1 and g_2 both belonging to G. I.e. a period is composed of the set of granules between g_1 and g_2, given that $g_1 \leq g_2$. The granules i_1 and i_2 are represented by two relative instants.

Concretely periods ($\boldsymbol{Per}$) contain:

- $\mathbf{ri}^-$ Start relative instant $(\text{ins}, \text{inv}) \in \boldsymbol{Ri}$
- $\mathbf{ri}^+$ End relative instant $(\text{ins}, \text{inv}) \in \boldsymbol{Ri}$

The following are examples of periods.

[2000-01-01$_{days}$, 2001-01-01$_{days}$]	The time between 2000-01-01 and 2001-01-01.
[1988-01-01$_{days}$, now_{days}- 01-00-00$_{days}$]	The time from 1988-01-01 to one year ago.
[1976-10-26$_{days}$, now_{days}]	The time from 1976-10-26 until now.
[now_{weeks}- 1_{weeks}, now_{weeks}]	The previous week.

A.2.5 Instant Container

The instant container data type is a multi set, IC, containing n instants $\{\text{ins}_1, \text{ins}_2 ... \text{ins}_n\}$.

A.2.6 Interval Container

Interval containers, IVC, contain n intervals $\{\text{inv}_1, \text{inv}_2 ... \text{inv}_n\}$.

A.2.7 Period Container

Finally period containers, PC contain n periods $\{\text{per}_1, \text{per}_2 ... \text{per}_n\}$.

A.3 Basic Operations on Temporal Data Types

Four semantics are possible for operations on temporal data of non-equal granularities and in all cases it may be necessary to convert between granularities. In the present semantic specification we leave it as an option to use any of the four semantics which are described shortly. For converting between granularities two operations are suggested [DELS98] : *scale* and *cast*. The two differ in the fact that scale may return indeterminate data, whereas cast always returns determinate data. The *scale* and *cast* operators are defined as follows:

$scale(g, H)$ The scale operator takes as input an instant $g = l_G \sim u_G$ in granularity G and another granularity H. It returns an instant $h = l_H \sim u_H$ in granularity H such that $l_G \sim u_G \subseteq l_H \sim u_H$. If no such instant exist an error is returned.

$cast(g, H)$ The cast operator is the determinate version of scale and is parameterized with an instant $g = l_G \sim u_G$ in granularity G and a granularity H. Cast returns an instant $h = l_H \sim u_H$ in H where $l_H \in min(scale(l_G, H))$ and $u_H \in min(scale(u_G, H))$. h is thus determinate if input g is determinate. The min function returns the smallest granule of the (possibly indeterminate) interval given.

Examples of the scale and cast functions for instants are

scale(2000_{years}, months)	$2000 - 01_{months} \sim 2000 - 12_{months}$
cast(2000_{years}, months)	$2000 - 01_{months}$
scale($cast$(2000_{days}, months))	$2000 - 01_{months}$

Similar semantics can be given for scaling and casting intervals, where an unanchored interval can be scaled to the indeterminate interval in a coarser granularity. Examples include

scale(1_{days}, years)	$0_{years} \sim 1_{years}$
cast(1_{days}, months)	0_{months}

Given two operands o_1 and o_2 from the set $\{\boldsymbol{Ins} \cup \boldsymbol{Inv}\}$, a binary operator/predicate $\odot \in \{>, <, =, +, -, \div\}$, and two granularities F and C that are *finer* respectively *minimally coarser* than G, we can express four semantics for operators:

Coarser semantics

$$o_1 \odot o_2 \begin{cases} scale(o_1, G_{o_2}).i \odot o_2.i & \text{if } G_{o_1} \trianglelefteq G_{o_2} \\ o_1.i \odot scale(o_2, G_{o_1}).i & \text{if } G_{o_1} \trianglerighteq G_{o_2} \\ scale(o_1, C).i \odot scale(o_2, C).i & \text{otherwise} \end{cases}$$

Finer semantics

$$o_1 \odot o_2 \begin{cases} o_2.i \odot scale(o_2, G_{o_1}).i & \text{if } G_{o_1} \trianglelefteq G_{o_2} \\ scale(o_1, G_{o_2}).i \odot o_2.i & \text{if } G_{o_1} \trianglerighteq G_{o_2} \\ scale(o_1, F).i \odot scale(o_2, F).i & \text{otherwise} \end{cases}$$

Right operand semantics

$$o_1 \odot o_2 = scale(o_1, G_{o_2}).i \odot o_2.i$$

Left operand semantics

$$o_1 \odot o_2 = o_1.i \odot scale(o_2, G_{o_1}).i$$

In the case of coarser semantics we scale the operand with the finest granularity to that of the other. If the granularities are not directly comparable we scale to a granularity that is minimally coarser than both operand-granularities (e.g. *weeks* and *months* will be scaled to *year*).

In the case of finer semantics the opposite is the case. Here we round the coarsest operand down to that of the finest or one that is finer than both.

Right (left) operand semantics scales the left (right) operand to that of the right (left) one.

Constructing and Converting the Data Types

When a function returns a result, the data type is constructed using the appropriate constructor operator. For our purpose of specifying a semantics it is sufficient to use a simple notation for such constructors. The notation is a tuple containing the elements of the data type in question. We add a subscript for each tuple for readability. The notation can be illustrated as follows.

Data type	**Notation**
instant	$(i,\ G)_{ins}$
interval	$(i,\ G)_{inv}$
relative instant	$(ins,\ inv)_{ri}$
period	$(ri,\ ri)_{per}$
instant container	$(ins_1, ins_2, \ldots, ins_n)_{ic}$
interval container	$(inv_1, inv_2, \ldots, inv_n)_{ivc}$
period container	$(per_1, per_2, \ldots, per_n)_{pc}$

Routines for converting between data types are specified with the prefix "to_", e.g. $ri.to\_ins()$, which creates an instant from a relative instant.

A.4 Operations of Temporal Types

This section contains a description of all operations available on the seven datatypes introduced above.

Notation is based on that mentioned in the previous section, such that for example "$ins > per.\mathrm{ri}^+$" would mean *ins* compared to the end instant of *per* using either of the comparison semantics.

A.4.1 Instant

An instant consists of the following operators.

We assume that $\mathrm{ri}^-, \mathrm{ri}^+, ri \in \boldsymbol{Ri}$; $G \in \mathbb{G}_{cal}$; $ins \in \boldsymbol{Ins}$, and $inv \in \boldsymbol{Inv}$

Syntax	**Ret**	**Semantics**
$ins.Granularity()$	gran	$ins.G$
$ins_1.Smaller(ins_2)$	bool	$ins_1 < ins_2$
$ins.Smaller(per)$	bool	$ins < per.\mathrm{ri}^-$
$ins_1.Greater(ins_2)$	bool	$ins_1 > ins_2$
$ins.Greater(per)$	bool	$ins > per.\mathrm{ri}^+$
$ins_1.Equal(ins_2)$	bool	$ins_1 = ins_2$
$ins_1.TotalyEqual(ins_1)$	bool	$ins_1 = ins_2 \wedge ins_1.G = ins_2.G$
$ins.Add(inv)$	ins	$ins + inv$
$ins.Add(per)$	ins	$ins + per.Duration()$
$ins.Sub(inv)$	ins	$ins - inv$
$ins.Sub(per)$	ins	$ins - per.Duration()$
$ins.Cast(G)$	ins	$cast(ins,\ G)$
$ri.to\_ins()$	ins	$(ri.\mathrm{ins}.Add(ri.\mathrm{inv}),\ ri.\mathrm{ins}.G)_{ins}$

A.4.2 Interval

An interval has the following operators.

We assume that $G \in \mathbb{G}_{cal}$; $inv \in \boldsymbol{Inv}$; and $i \in \mathbb{N}$

Syntax	**Ret**	**Semantics**
$inv_1.Granularity()$	gran	$inv_1.\lambda$
$inv_1.Smaller(inv_2)$	bool	$inv_1 < inv_2$
$inv.Smaller(per)$	bool	$inv < per.Duration()$
$inv_1.Greater(inv_2)$	bool	$inv_1 > inv_2$
$inv_1.Equal(inv_2)$	bool	$inv_1 = inv_2$
$inv_1.TotalyEqual(inv_2)$	bool	$inv_1 = inv_2 \wedge inv_1.\lambda = inv_2.\lambda$
$inv.Neg()$	inv	$-inv.i$
$inv_1.Abs()$	inv	$\begin{cases} inv_1 & inv_1.i > 0 \\ inv_1.Neg() & \text{Otherwise.} \end{cases}$
$inv_1.Sub(inv_2)$	inv	$inv_1 - inv_2$
$inv.Sub(per)$	inv	$inv - per.Duration()$
$inv_1.Add(inv_2)$	inv	$inv_1 + inv_2$
$inv.Add(per)$	inv	$inv + per.Duration()$
$inv_1.Div(inv_2)$	inv	$\frac{inv_1}{inv_2}$
$inv.Div(per)$	inv	$\frac{inv}{per.Duration()}$
$inv.Cast(G)$	inv	$cast(inv,\ G)$

A.4.3 Relative Instant

Assuming $G \in \mathbb{G}_{cal}$; $inv \in \boldsymbol{Inv}$; $\mathrm{ri}^-, \mathrm{ri}^+, ri \in \boldsymbol{Ri}$; $ins \in \boldsymbol{Ins}$; and $per \in \boldsymbol{Per}$ we specify the operators of relative instants below.

Syntax	**Ret**	**Semantics**
$ri.Granularity()$	gran	$ri.ins.Granularity()$
$ri_1.Smaller(ri_2)$	bool	$ri_1.to\_ins().Smaller(ri_2.to\_ins())$
$ri.Smaller(ins)$	bool	$ri.to\_ins().Smaller(ins)$
$ri.Smaller(per)$	bool	$ri.to\_ins().Smaller(per.\mathrm{ri}^-)$
$ri_1.Greater(ri_2)$	bool	$ri_1.to\_ins().Greater(ri_2.to\_ins())$
$ri.Greater(ins)$	bool	$ri.to\_ins().Greater(ins)$
$ri.Greater(per)$	bool	$ri.to\_ins().Greater(per.\mathrm{ri}^+)$
$ri_1.Equal(ri_2)$	bool	$ri_1.to\_ins().Equal(ri_2.to\_ins())$
$ri.Equal(ins)$	bool	$ri.to\_ins().Equal(ins)$
$ri_1.TotalEqual(ri_2)$	bool	$ri_1.to\_ins().TotalEqual(ri_2.to\_ins()) \wedge ri_1.inv(TotalEqual(ri_2.inv)$
$ri.Add(inv)$	ins	$(\mathrm{ins},\ ri.\mathrm{inv}.Add(inv))_{ri}$
$ri.Sub(inv)$	ins	$(\mathrm{ins},\ ri.\mathrm{inv}.Sub(inv))_{ri}$
$ri.Add(per)$	ins	$(\mathrm{ins},\ ri.\mathrm{inv}.Add(per.Duration())_{ri}$
$ri.Sub(per)$	ins	$(\mathrm{ins},\ ri.\mathrm{inv}.Sub(per.Duration())_{ri}$
$ri.to\_ins()$	ins	$\begin{cases} \text{ins} & \text{if } ri \text{ is of type } normal \\ (\text{current system time})_{ins} & \text{if } ri \text{ is of type } now \end{cases}$
$ri.Cast(G)$	ins	$(ins.Cast(G), inv.Cast(G))_{ri}$

A.4.4 Period

The operators of the period data type is as follows:

We assume that $G \in \mathbb{G}_{cal}$; $ins \in \boldsymbol{Ins}$; $inv \in \boldsymbol{Inv}$; $\mathrm{ri}^-, \mathrm{ri}^+, ri \in \boldsymbol{Ri}$; and $per \in \boldsymbol{Per}$

Syntax	**Ret**	**Semantics**
$per.Granularity()$	gran	$per.\mathrm{ri}^-.G$
$per_1.TotalEqual(per_2)$	bool	$per_1.\mathrm{ri}^-.TotalEqual(per_2.\mathrm{ri}^-) \ \wedge\ per_1.\mathrm{ri}^+.TotalEqual(per_2.\mathrm{ri}^+)$
$per.Add(inv)$	per	$(\mathrm{ri}^-, \mathrm{ri}^+.Add(inv))_{per}$
$per_1.Add(per_2)$	per	$per_1.Add(per_2.Duration())$
$per.Sub(inv)$	per	$(\mathrm{ri}^-, \mathrm{ri}^+.Sub(inv))_{per}$
$per_1.Sub(per_2)$	per	$per_1.Sub(per_2.Duration())$
$per.Move(inv)$	per	$(\mathrm{ri}^-.Add(inv),\ \mathrm{ri}^+.Add(inv))_{per}$
$per.Move(per)$	per	$(\mathrm{ri}^-.Add(per),\ \mathrm{ri}^+.Add(per))_{per}$
$per.Duration()$	inv	$per.\mathrm{ri}^+.Sub(per.\mathrm{ri}^-)$
$per.Smaller(inv)$	bool	$per.Duration().Smaller(inv)$
$per.Greater(inv)$	bool	$per.Duration().Greater(inv)$
$per.Equal(inv)$	bool	$per.Duration().Equal(inv)$
$per_1.Equal(per_2)$	bool	$per_1.\mathrm{ri}^+.Equals(per_2.\mathrm{ri}^+) \wedge per_1.\mathrm{ri}^-.Equals(per_2.\mathrm{ri}^-)$
$per_1.Intersect(per_2)$	per	$(max(per_1.\mathrm{ri}^-,\ per_2.\mathrm{ri}^-),\ min(per_1.\mathrm{ri}^+,\ per_2.\mathrm{ri}^+))_{per}$
$per_1.Contains(per_2)$	bool	$per_2.\mathrm{ri}^-.Greater(per_1.\mathrm{ri}^-) \wedge per_2.\mathrm{ri}^+.Smaller(per_1.\mathrm{ri}^+)$
$per.Contains(ins)$	bool	$per.\mathrm{ri}^-.Smaller(ins) \wedge per_2.\mathrm{ri}^+.Greater(ins)$
$per.Contains(ri)$	bool	$per.\mathrm{ri}^-.Smaller(ri.to\_ins()) \wedge per_2.\mathrm{ri}^+.Greater(ri.to\_ins())$
$per_1.RightOverlaps(per_2)$	bool	$per_1.\mathrm{ri}^+.Greater(per_2.\mathrm{ri}^+) \vee per_1.\mathrm{ri}^-.Smaller(per_2.\mathrm{ri}^+)$
$per_1.LeftOverlaps(per_2)$	bool	$per_1.\mathrm{ri}^+.Greater(per_2.\mathrm{ri}^-) \vee per_1.\mathrm{ri}^-.Smaller(per_2.\mathrm{ri}^-)$
$per_1.StartsInside(per_2)$	bool	$per_1.\mathrm{ri}^-.Greater(per_2.\mathrm{ri}^-) \vee per_1.\mathrm{ri}^-.Smaller(per_2.\mathrm{ri}^+)$
$per_1.EndsInside(per_2)$	bool	$per_1.\mathrm{ri}^+.Greater(per_2.\mathrm{ri}^-) \vee per_1.\mathrm{ri}^+.Smaller(per_2.\mathrm{ri}^+)$
$per_1.Overlaps(per_2)$	bool	$per_1.\mathrm{ri}^+.Greater(per_2.\mathrm{ri}^-) \vee per_1.\mathrm{ri}^-.Smaller(per_2.\mathrm{ri}^+)$
$per.Overlaps(ins)$	bool	$per.\mathrm{ri}^-.Smaller(ins) \wedge per_2.\mathrm{ri}^+.Greater(ins)$
$per.Overlaps(ri)$	bool	$per.\mathrm{ri}^-.Smaller(ri.to\_ins()) \wedge per_2.\mathrm{ri}^+.Grater(ri.to\_ins())$
$per_1.Meets(per_2)$	bool	$per_1.\mathrm{ri}^-.Equal(per_2.\mathrm{ri}^+) \vee per_1.\mathrm{ri}^+.Equal(per_2.\mathrm{ri}^-)$
$per.Meets(ins)$	bool	$per.\mathrm{ri}^-.Equal(ins) \vee per.\mathrm{ri}^+.Equal(ins)$
$per.Meets(ri)$	bool	$per.\mathrm{ri}^-.Equal(ri.to\_ins()) \vee per.\mathrm{ri}^+.Equal(ri.to\_ins())$
$per_1.Precedes(per_2)$	bool	$per_1.\mathrm{ri}^+.Smaller(per_2.\mathrm{ri}^-)$
$per.Precedes(ins)$	bool	$per_1.\mathrm{ri}^+.Smaller(ins)$
$per.Precedes(ri)$ df	bool	$per_1.\mathrm{ri}^+.Smaller(ri.to\_ins())$
$per_1.Succedes(per_2)$	bool	$per_1.\mathrm{ri}^-.Greater(per_2.\mathrm{ri}^+)$
$per.Succedes(ins)$	bool	$per_1.\mathrm{ri}^-.Greater(ins)$
$per.Succedes(ri)$	bool	$per_1.\mathrm{ri}^-.Greater(ri.to\_ins())$
$per.Cast(G)$	per	$(per_1.\mathrm{ri}^-.Cast(G),\ per_1.\mathrm{ri}^+.Cast(G)_{per}$

A.4.5 Instant Container

The semantics of the available operators on instant containers are listed below. First operators which are also found on the instant data type is listed. Following, set operators applicable to instant containers are listed.

$G \in \mathbb{G}_{cal}$, $ins \in \boldsymbol{Ins}$, $inv \in \boldsymbol{Inv}$, $\mathrm{ri}^-, \mathrm{ri}^+, ri \in \boldsymbol{Ri}$, $per \in \boldsymbol{Per}$, $n \in \mathbb{N}$

Instant Container Semantics

Syntax	**Ret**	**Semantics**
$IC.Granularity()$	gran	$\begin{cases} ins.Granularity() \text{ where } ins \in PC & \text{if granularity is homogeneous in the container} \\ error & \text{otherwise} \end{cases}$
$IC_1.Smaller(IC_2)$	IC	$\{ins' \mid IC.Contains(ins') \wedge ins'.Smaller(IC.Smallest())\}$
$IC.Smaller(ins)$	IC	$\{ins' \mid IC.Contains(ins') \wedge ins'.Smaller(ins)\}$
$IC.Smaller(ri)$	IC	$\{ins' \mid IC.Contains(ins') \wedge ins'.Smaller(to\_ins(ri))\}$
$IC.Smaller(per)$	IC	$\{ins' \mid IC.Contains(ins') \wedge ins'.Smaller(per.\mathrm{ri}^-)\}$
$IC_1.Greater(IC_2)$	IC	$\{ins' \mid IC.Contains(ins') \wedge ins'.Greater(IC.Greatest())\}$
$IC.Greater(ins)$	IC	$\{ins' \mid IC.Contains(ins') \wedge ins'.Greater(ins)\}$
$IC.Greater(ri)$	IC	$\{ins' \mid IC.Contains(ins') \wedge ins'.Greater(to\_ins(ri))\}$
$IC.Greater(per)$	IC	$\{ins' \mid IC.Contains(ins') \wedge ins'.Greater(per.\mathrm{ri}^+)\}$
$IC_1.Equal(IC_2)$	IC	$\{ins' \mid IC_1.Contains(ins') \wedge IC_2.Contains(ins')\}$
$IC.Equal(ins)$	IC	$\{ins' \mid IC_1.Contains(ins') \wedge ins'.Equal(ins)\}$
$IC.Equal(ri)$	IC	$\{ins' \mid IC_1.Contains(ins') \wedge ins'.Equal(to\_ins(ri))\}$
$IC_1.TotalEqual(IC_2)$	bool	$\forall ins'(ins' \in IC_2 \wedge IC_1.Contains(ins')) \wedge$ $\forall ins(ins \in IC_1 \wedge IC_2.Contains(ins))$
$IC.Add(ins)$	IC	$(ins_1.Add(ins), ins_2.Add(ins), \ldots, ins_n.Add(ins))_{IC}$
$IC.Add(per)$	IC	$(ins_1.Add(per), ins_2.Add(per), \ldots, ins_n.Add(per))_{IC}$
$IC.Sub(ins)$	IC	$(ins_1.Sub(ins), ins_2.Sub(ins), \ldots, ins_n.Sub(ins))_{IC}$
$IC.Sub(inv)$	IC	$(ins_1.Sub(inv), ins_2.Sub(inv), \ldots, ins_n.Sub(inv))_{IC}$
$IC.Sub(per)$	IC	$(ins_1.Sub(per), ins_2.Sub(per), \ldots, ins_n.Sub(per))_{IC}$
$IC_1.Intersect(IC_2)$	IC	$\{ins \mid IC_1.Contains(ins) \wedge IC_2.Contains(ins)\}$
$IC_1.Intersect(ins)$	IC	$\{ins \mid IC_1.Contains(ins) \wedge IC_2.Contains(ins)\}$
$IC_1.Intersect(ri)$	IC	$\{ins \mid IC_1.Contains(ins) \wedge IC_2.Contains(ins)\}$
$IC_1.Intersect(per)$	IC	$\{ins \mid IC_1.Contains(ins) \wedge IC_2.Contains(ins)\}$
$IC_1.Contains(IC_2)$	bool	$\forall ins'(IC_2.Contains(ins') \wedge IC_1.Contains(ins'))$
$IC.Contains.(ins)$	bool	$\exists ins'(IC.Contains(ins') \wedge ins.Equal(ins'))$
$IC.Contains(ri)$	bool	$\exists ins'(IC.Contains(ins') \wedge to\_ins(ri).Equal(ins'))$
$IC.Contains(per)$	bool	$\exists ins'(IC.Contains(ins') \wedge per.Contains(ins'))$
$IC_1.Overlaps(IC_2)$	bool	$IC_1.Contains(IC_2)$
$IC.Overlaps(ins)$	bool	$IC_1.Contains(ins)$
$IC.Overlaps(ri)$	bool	$IC_1.Contains(ri)$
$IC.Overlaps(per)$	bool	$IC_1.Contains(per)$
$IC.Greatest()$	ins	$\{ins \mid ins \in IC \wedge \neg\exists ins'(IC.Contains(ins') \wedge ins'.Smaller(ins))$
$IC.Smallest()$	ins	$\{ins \mid ins \in IC \wedge \neg\exists ins'(IC.Contains(ins') \wedge ins'.Greater(ins))$
$IC.Count()$	num	n
$IC.Duplicates()$	bool	$\exists ins \exists ins'(IC.Contains(ins) \wedge IC.Contains(ins') \wedge ins.Equal(ins'))$

Instant Container Semantics

$IC.Coalesce()$	IC	Returns the coalesced version of IC, i.e. where duplicate instances have been combined into just one instant.
$IC.Cast(G)$	IC	$(ins_1.Cast(G), ins_2.Cast(G), \ldots , ins_n.Cast(G))_{IC}$
$IC.AddInstant(ins)$	IC	$\{ins' \mid IC.Contains(ins') \vee ins'.Equals(ins)\}$
$IC.AddInstant(ri)$	IC	$\{ins' \mid IC.Contains(ins') \vee ins'.Equals(to\_ins(ri))\}$
$IC.RemoveInstant(ins)$	IC	$\{ins' \mid IC.Contains(ins') \wedge \neg ins'.Equals(ins)\}$
$IC.RemoveInstant(ri)$	IC	$\{ins' \mid IC.Contains(ins') \wedge \neg ins'.Equals(to\_ins(ri))\}$

A.4.6 Interval Container

The semantics of various operations on interval containers is given below. As in the case of instant containers above, we first specify operations inherited from intervals, then set and other operations.

Given $G \in \mathbb{G}_{cal}$; $inv \in \boldsymbol{Inv}$; $per \in \boldsymbol{Per}$; and $n \in \mathbb{N}$ we have the following operations on interval containers.

Interval Container Semantics

Syntax	**Ret**	**Semantics**
$IVC.Granularity()$	gran	$\begin{cases} inv.Granularity() \text{ where } inv \in IVC & \text{if granularity is homogeneous in the container} \\ error & \text{otherwise} \end{cases}$
$IVC_1.Smaller(IVC_2)$	IVC	$\{inv' \mid IVC.Contains(inv') \ \wedge \ ins'.Smaller(IVC.Smallest())\}$
$IVC.Smaller(inv)$	IVC	$\{inv' \mid IVC.Contains(inv') \ \wedge \ ins'.Smaller(inv))\}$
$IVC.Smaller(per)$	IVC	$\{inv' \mid IVC.Contains(inv') \ \wedge \ ins'.Smaller(per.Duration()))\}$
$IVC_1.Greater(IVC_2)$	IVC	$\{inv' \mid IVC.Contains(inv') \ \wedge \ ins'.Greater(IVC.Greatest())\}$
$IVC.Greater(inv)$	IVC	$\{inv' \mid IVC.Contains(inv') \ \wedge \ ins'.Greater(inv))\}$
$IVC.Greater(per)$	IVC	$\{inv' \mid IVC.Contains(inv') \ \wedge \ ins'.Greater(per.Duration()))\}$
$IVC_1.Equal(IVC_2)$	IVC	$\{inv' \mid IVC_1.Contains(inv') \ \wedge \ IVC_2.Contains(inv')\}$
$IVC.Equal(inv)$	IVC	$\{inv' \mid IVC_1.Contains(inv') \ \wedge \ ins'.Equal(inv)\}$
$IVC.Equal(per)$	IVC	$\{inv' \mid IVC_1.Contains(inv') \ \wedge \ ins'.Equal(per.Duration())\}$
$IVC_1.TotalEqual(IVC_2)$	bool	$\forall inv'(IVC_2.Contains(inv') \Rightarrow IVC_1.Contains(inv')) \wedge$ $\forall inv(IVC_1.Contains(inv) \Rightarrow IVC_2.Contains(inv))$
$IVC.Sub(inv)$	IVC	$\{inv' \mid \exists inv''(IVC.Contains(inv'') \wedge inv''.Sub(inv).Equals(inv'))\}$
$IVC.Sub(per)$	IVC	$\{inv' \mid \exists inv''(IVC.Contains(inv'') \wedge inv''.Sub(per).Equals(inv'))\}$
$IVC.Add(inv)$	IVC	$\{inv' \mid \exists inv''(IVC.Contains(inv'') \wedge inv''.Add(inv).Equals(inv'))\}$
$IVC.Add(per)$	IVC	$\{inv' \mid \exists inv''(IVC.Contains(inv'') \wedge inv''.Add(per).Equals(inv'))\}$
$IVC.Div(inv)$	IVC	$\{inv' \mid \exists inv''(IVC.Contains(inv'') \wedge inv''.Div(inv).Equals(inv'))\}$
$IVC.Div(per)$	IVC	$\{inv' \mid \exists inv''(IVC.Contains(inv'') \wedge inv''.Div(per).Equals(inv'))\}$
$IVC.Neg()$	IVC	$\{inv' \mid \exists inv(IVC.Contains(inv) \wedge inv'.Equals(inv.Neg()))\}$
$IVC.Abs()$	IVC	$\{inv' \mid \exists inv(IVC.Contains(inv) \wedge inv'.Equals(inv.Abs()))\}$
$IVC_2.Intersect(IVC_2)$	IVC	$\{inv' \mid IVC_1.Contains(inv') \wedge IVC_2.Contains(inv')\}$
$IVC.Intersect(inv)$	IVC	$IVC.Equal(inv)$
$IVC.Intersect(per)$	IVC	$IVC.Equal(per)$
$IVC_1.Contains(IVC_2)$	bool	$\forall inv'(IVC_1.Contains(inv') \wedge IVC_2.Contains(inv'))$
$IVC.Contains(inv)$	bool	$\exists inv'(IVC.Contains(inv') \ \wedge \ inv.Equal(inv'))$
$IVC.Contains(per)$	bool	$\exists inv'(IVC.Contains(inv') \ \wedge \ per.Duration().Equal(inv'))$
$IVC_1.Overlaps(IVC_2)$	bool	$IVC_1.Contains(IVC_2)$
$IVC.Overlaps(inv)$	bool	$IVC_1.Contains(inv)$
$IVC.Overlaps(per)$	bool	$IVC_1.Contains(per)$
$IVC.Greatest()$	inv	$\{inv \mid IVC.Contains(inv) \wedge \neg\exists inv'(IVC.Contains(inv') \wedge inv'.Greater(inv))\}$
$IVC.Smallest()$	inv	$\{inv \mid IVC.Contains(inv) \wedge \neg\exists inv'(IVC.Contains(inv') \wedge inv'.Smaller(inv))\}$
$IVC.Count()$	num	n
$IVC.Duplicates()$	bool	$\exists inv, inv'(IVC.Contains(inv) \wedge IVC.Contains(inv') \wedge inv.Equals(inv'))$

Interval Container Semantics		
$IC.Coalesce()$	IC	Returns the coalesced version of IVC, i.e. where duplicate instances have been combined into just one interval of the given size and granularity.
$IVC.Cast(G)$	IVC	$(inv_1.Cast(G),\ inv_2.Cast(G),\ \ldots,inv_n.Cast(G))_{IVC}$
$IVC.AddInterval(inv)$	IVC	$\{inv' \mid IVC.Contains(inv') \vee inv'.Equals(inv)\}$
$IVC.AddInterval(per)$	IVC	$\{inv' \mid IVC.Contains(inv') \vee inv'.Equals(per.Duration())\}$
$IVC.RemoveInterval(inv)$	IVC	$\{inv' \mid IVC.Contains(inv') \wedge \neg inv'.Equals(inv)\}$
$IVC.RemoveInterval(per)$	IVC	$\{inv' \mid IVC.Contains(inv') \wedge \neg inv'.Equals(per.Duration())\}$

A.4.7 Period Container

The semantics of a period container is listed below, once again with period operations first, then set operations.

Given $G \in \mathbb{G}_{cal}$; $ins \in \boldsymbol{Ins}$; $inv \in \boldsymbol{Inv}$; $\mathrm{ri}^{-}, \mathrm{ri}^{+}, ri \in \boldsymbol{Ri}$; $per \in \boldsymbol{Per}$; and $n \in \mathbb{N}$ we specify the following operations.

Period Container Semantics

Syntax	**Ret**	**Semantics**
$PC.Granularity()$	gran	$\begin{cases} per.Granularity() \text{ where } per \in PC & \text{if granularity is homogeneous in the container} \\ error & \text{otherwise} \end{cases}$
$PC_1.Smaller(PC_2)$	PC	$\{per' \mid PC_1.Contains(per') \ \wedge \ per'.Duration().Smaller(PC_2.Smallest().Duration())\}$
$PC.Smaller(inv)$	PC	$\{per' \mid PC.Contains(per') \ \wedge \ per'.Duration().Smaller(inv)\}$
$PC_1.Greater(PC_2)$	PC	$\{per' \mid PC_1.Contains(per') \ \wedge \ per'.Duration().Greater().(PC_2.Greatest().Duration())\}$
$PC.Greater(inv)$	PC	$\{per' \mid PC.Contains(per') \ \wedge \ per'.Duration().Greater(inv)\}$
$PC_1.Equal(PC_2)$	PC	$\{per' \mid PC_1.Contains(per') \ \wedge \ PC_2.Contains(per')\}$
$PC.Equal(inv)$	PC	$\{per' \mid PC_1.Contains(per') \ \wedge \ per'.Duration().Equal(inv)\}$
$PC.Equal(per)$	PC	$\{per' \mid PC_1.Contains(per') \ \wedge \ per'.Equal(per)\}$
$PC_1.TotalEqual(PC_2)$	bool	$\forall per'(PC_2.Contains(per') \wedge \exists per''(PC_1.Contains(per'') \wedge per''.Equals(per'))) \wedge \ \forall \ per(PC_1.Contains(per) \ \wedge \ \exists per'''(PC_1.Contains(per''') \ \wedge \ per'''.Equals(per))$
$PC.Sub(inv)$	PC	$\{per' \mid \exists per''(PC.Contains(per'') \wedge per''.Sub(inv).Equals(per')\}$
$PC.Sub(per)$	PC	$\{per' \mid \exists per''(PC.Contains(per'') \wedge per''.Sub(per).Equals(per')\}$
$PC.Add(inv)$	PC	$\{per' \mid \exists per''(PC.Contains(per'') \wedge per''.Add(inv).Equals(per')\}$
$PC.Add(per)$	PC	$\{per' \mid \exists per''(PC.Contains(per'') \wedge per''.Add(per).Equals(per')\}$
$PC.Durations()$	IVC	$\{inv' \mid \exists per(PC.Contains(per) \wedge per.Duration.Equals(inv'))\}$
$PC.Move(inv)$	PC	$\{per' \mid \exists per(PC.Contains(per) \wedge per.Move(inv).Equals(per'))\}$
$PC.Move(per)$	PC	$\{per' \mid \exists per(PC.Contains(per) \wedge per.Move(per).Equals(per'))\}$
$PC_1.Overlaps(PC_2)$	bool	$\forall per \forall per'(PC_1.Contains(per) \wedge PC_2.Contains(per') \wedge per.Overlaps(per'))$
$PC.Overlaps(per)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Overlaps(per))$
$PC.Overlaps(ins)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Overlaps(ins))$
$PC.Overlaps(ri)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Overlaps(ri))$
$PC_1.Meets(PC_2)$	bool	$\forall per \forall per'(PC_1.Contains(per) \wedge PC_2.Contains(per') \wedge per.Meets(per'))$
$PC.Meets(per)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Meets(per))$
$PC.Meets(ins)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Meets(ins))$
$PC.Meets(ri)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Meets(ri))$
$PC_1.Precedes(PC_2)$	bool	$\forall per \forall per'(PC_1.Contains(per) \wedge PC_2.Contains(per') \wedge per.Precedes(per'))$
$PC.Precedes(per)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Precedes(per))$
$PC.Precedes(ins)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Precedes(ins))$
$PC.Precedes(ri)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Precedes(ri))$
$PC_1.Succeeds(PC_2)$	bool	$\forall per \forall per'(PC_1.Contains(per) \wedge PC_2.Contains(per') \wedge per.Succeeds(per'))$
$PC.Succeeds(per)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Succeeds(per))$
$PC.Succeeds(ins)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Succeeds(ins))$
$PC.Succeeds(ri)$	bool	$\exists per'(PC_1.Contains(per') \wedge per'.Succeeds(ri))$

Period Container Semantics

$PC.Cast(G)$	PC	$(per_1.Cast(G), per_2.Cast(G), \ldots, per_n.Cast(G))_{PC}$
$PC.Remove(inv)$	PC	$\{per' \mid PC.Contains(per') \land \neg per'.Equals(per)\}$
$PC.Remove(per)$	PC	$\{per' \mid PC.Contains(per') \land \neg per'.Equals(per)\}$
$PC_1.Contains(PC_2)$	bool	$\forall per'(PC_2.Contains(per') \land PC_1.Contains(per'))$
$PC.Contains(per)$	bool	$\exists per'(PC.Contains(per') \land per'.Equals(per))$
$PC.Contains(inv)$	bool	$\exists inv'(PC.Contains(per') \land inv'.Duration().Equals(inv))$
$PC_2.Intersect(PC_2)$	PC	$\{per \mid PC_1.Contains(per) \land PC_2.Contains(per)\}$
$PC_1.Union(PC_2)$	PC	$\{per \mid PC_1.Contains(per) \lor PC_2.Contains(per)\}$
$PC.Largest()$	PC	$\{per \mid PC.Contains(per) \land \neg\exists per'(PC.Contains(per') \land per'.Succedes(per))\}$
$PC.Smallest()$	PC	$\{per \mid PC.Contains(per) \land \neg\exists per'(PC.Contains(per') \land per'.Precedes(per))\}$
$PC.Count()$	num	n
$PC.Duplicates()$	bool	$\exists per \exists per'(PC.Contains(per) \ \land \ PC.Contains(per') \ \land \ per.Equals(per'))$
$PC.Coalesce()$	PC	Returns PC' which is the coalesced version of PC, i.e. all duplicates have been merged into just one single period
$PC.TimeSlice(ins)$	bool	$\exists per(PC.Contains(per) \land per.Contains(ins))$
$PC.TimeSlice(per)$	PC	$\{per' \mid \exists per''(PC.Contains(per'') \land per''.Overlaps(per) \land per'.Equals(per.Intersect(per'')))\}$
$PC.Cover()$	per	$(PC.Smallest.\mathrm{ri}^-,\ PC.Largest().\mathrm{ri}^+)_{per}$

Bibliography

[BBM$^+$99] M. Böhlen, L. Bukauskas, R. Marti, R. T. Snodgrass, and C. S. Jensen. Tiger, 1999. Implementation of Tiger can be downloaded from the Tiger web pages at URL: http://www.cs.auc.dk/ tiger.

[BBS98] M. Böhlen, R. Busatto, and C. S.Jensen. Point versus Interval-based Temporal Data Models. Technical report, TimeCenter, January 1998.

[BJ96] M. H. Böhlen and C. S. Jensen. A Seamless Integration of Time into SQL. Technical report, Technical Report R-96–2049, Aalborg University, Department of Computer Science, Frederik Bajers Vej 7E, DK–9220 Aalborg Øst, Denmark, December 1996.

[BJ97] M. H. Böhlen and C. . Jensen. Temporal Statement Modifiers. Available via http://www.cs.auc.dk/research/DBS/teaching/DAT5E99/tdb2.ps.gz, 1997.

[BJSS95] M. Böhlen, C. S. Jensen, A. Steiner, and R. Snodgrass. Implementation of TimeDB can be downloaded at URL: http://www.iesd.auc.dk/general/DBS/tdb/TimeCenter/Software/TimeDB.tar.gz, 1995.

[BJSS98] R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas. Light-Weight Indexing of General Bitemporal Data. Technical report, TimeCenter, September 1998.

[Böh95] M. Böhlen. Temporal Database System Implementations. *ACM SIGMOD Record*, 24(4), December 1995.

[BSS97] M. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. Technical report, TimeCenter, April 1997.

[BSSJ98] R. Bliujute, S. Saltenis, G. Slivinskas, and C. S. Jensen. Developing a DataBlade for a New Index. Technical report, TimeCenter, September 1998.

[Coo99] Oracle Coorporation. Oracle8i Spatial. http://technet.oracle.com/doc.pdf/inter.815/a67295.pdf, February 1999.

[Dav00] Judith R. Davis. Ibm db2 universal database : Building extensible, scalable business solutions. IBM Coorporation, http://www-4.ibm.com/software/data/pubs/papers/db2udb/db2udb.pdf, Feb 2000.

[dBS96] Jochen Van den Bercken and Bernhard Seeger. Query processing techniques for multiversion access methods. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 168–179. Morgan Kaufmann, 1996.

[DELS98] C. E. Dyreson, W. S. Evans, H. Lin, and R. T. Snodgrass. Efficiently Supporting Temporal Granularities. Technical report, TimeCenter, 1998.

[DLM97] B. Daniell, J. Leland, and D. Maneval. INFORMIX Universal Server, DataBlade API Programmer's manual, June 1997. Available online from http://www.informix.com/answers/english/docs/912ius/4115.pdf.

[DS91] C. E. Dyreson and R. T. Snodgrass. Temporal Indeterminacy. Technical Report TR 91-30, University of Arizona Department of Computer Science, December 1991.

[DS94] C. E. Dyreson and R. T. Snodgrass. Temporal Granularity and Indeterminacy: Two Sides of the Same Coin. Technical Report TR 94-06, uazcsd, Feb. 1994.

[DSJ93] C. E. Dyreson, R. T. Snodgrass, and C. S. Jensen. On the Semantics of "Now" in Temporal Databases. TempIS Technical Report 42, University of Arizona Department of Computer Science, April 1993.

[FP97] S. Feuerstein and B. Pribyl. *Oracle PL/SQL programming.* O'Reilly & Associates, Inc., second edition, 1997.

[GJ97] H. Gregersen and C. S. Jensen. Temporal Entity-Relationship Models - a Survey. Technical report, TimeCenter, January 1997.

[Gut99] R. Gutman. Space-Filling Curves in Geospatial Applications. Dr. Dobbs Journal, July 1999.

[Je98] C. S. Jensen and C. E. Dyreson [eds]. A Consensus Glossary of Temporal Database Concepts. In *Temporal Databases: Research and Practice. (the book grow out of a Dagstuhl Seminar, June 23-27, 1997)*, number 1, pages 367–405. Springer, February 1998.

[Jen99] C. S. Jensen. *Temporal Database Management.* August 1999. http://www.cs.auc.dk/c̃sj/Thesis/.

[KS95] N. Kline and R.T. Snodgrass. Computing Temporal Aggregates. In *Proceedings of the IEEE International Conference on Database Engineering, 1995*, Tapei, Taiwan, March 1995.

[LKC99] J. Lee, D. Kim, and C. Chung. Multi-dimensional Selectivity Estimation Using Compressed Histogram Information. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 205–214. ACM Press, 1999.

[LO99] D. Lorentz and D. Oertel. *Oracle8i SQL Reference, release 8.1.5.* Oracle Coorporation, February 1999.

[LZ98] M. S. Lakshmi and S. Zhou. Selectivity Estimation in Extensible Databases - A Neural Network Approach. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 623–627. Morgan Kaufmann, 1998.

[MCS88] M. V. Mannino, P. Chu, and T. Sager. Statistical Profile Estimation in Database Systems. *Computing Surveys*, 20(3):191–221, 1988.

[ME00] J. Melton and A. Eisenberg. SQL Standardization: The Next Steps. *SIGMOD Record*, 29(1), March 2000.

[Mel96] J. Melton. SQL/Temporal. ISO/IEC JTC1/SC 21/WG 3 DBL-MCI-0012, July 1996.

[MLI99] B. Moon, I. F. V. López, and V. Immanuel. Scalable Algorithms for Large Temporal Aggregation. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia.* IEEE Computer Society, 1999.

[ND98] M. A. Nascimento and M. H. Dunham. Indexing Valid Time Databases Via B+-trees - The MAP21 Approach. Technical report, TimeCenter, March 1998.

[PI97] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 486–495. Morgan Kaufmann, 1997.

[PJ98] D. Pfoser and C. S. Jensen. Incremental Join of Time-Oriented Data. Technical report, TimeCenter, September 1998.

[RCG$^+$99] D. Raphaely, M. Cyran, J. Gibb, V. Krishnamurthy, M. Krishnaprasad, J. Melnick, and R. Urbano R. Smith. *Application Developer's Guide - Fundamentals.* Oracle Coorporation, release 8.1.5 edition, Feburay 1999.

[RP92] J. F. Roddick and J. D. Patrick. Temporal Semantics in Information Systems – A Survey. *Information Systems*, 17(3):249–267, October 1992.

[RRM99] D. Raphaely, J. Rawles, and C. Murray. *Oracle8i Data Cartridge Developer's guide.* Oracle Coorporation, release 2 (8.1.6) edition, December 1999.

[RS87] L. Rowe and M. Stonebraker. The postgres papers. Technical Report UCB/ERL M86/85, University of California, Berkeley, CA, June 1987.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187–260, 1984.

[SKS97] A. Silberschatz, H. F. Korth, and S. Sudarsahn. *Database System Concepts.* McGraw-Hill, third edition, 1997.

[SN98] J. R. O. Silva and M. A. Nascimento. An Incremental Index for Bitemporal Databases. Technical report, TimeCenter, November 1998.

[Sno95] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Number 0-7923-9614-6. Kluwer Academic Publishers, 1995.

[Sno00] R. Snodgrass. *Developing Time-Oriented Applications in SQL*. Morgan Kaufmann, 2000.

[SS94] R. T. Snodgrass and M. Soo. Supporting Multiple Calendars in TSQL2: An Overview. commentary, TSQL2 Design Committee, September 1994.

[TGJ99] K. Thrysøe, B. Gundersen, and T. Jørgensen. Optimizing Algorithms for Temporal Set Difference. May 1999.

[Thr00] K. Thrysøe. Extending the Oracle8i ORDBMS for Temporal Data. PostScript available from http://www.cs.auc.dk/s̃uaq/extendingOracle.ps, January 2000.

[TJS98] K. Torp, C. S. Jensen, and R. T. Snodgrass. Stratum Approaches to Temporal DBMS Implemenations. In Jianhua Shao Barry Eaglestone, Bipin C. Desai, editor, *Proceedings of the 1998 International Database Engineering and Applications Symposium, Cardiff, Wales, U.K., July 8-10, 1998*, pages 4–13. IEEE Computer Society, 1998.

[Wal92] R. M. Wald. *Space, Time and Gravity: the Theory of the Big Bang and Black Holes.* University of Chicago, 2nd edition, 1992.

[WJW98] Y. Wu, S. Jajodia, and X. S. Wang. Temporal Database Bibliography Update. In *Temporal Databases: Research and Practice. (the book grow out of a Dagstuhl Seminar, June 23-27, 1997)*, pages 338–366. Springer, 1998.

[YC91] C. Yau and G. S. W. Chat. TempSQL – A Language Interface to a Temporal Relational Model. *Information Sc. & Tech.*, pages 44–60, October 1991.

[YWY99] J. Yang, J. Widom, and H. C. Ying. TIP: A Temporal Extension to Informix. Available via http://www-db.stanford.edu/pub/papers/yyw-tipdemp.ps. Demonstration description., October 1999.