University of California
Los Angeles

Data Models and Query Languages of Spatio-Temporal Information

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Cindy Xinmin Chen

2001

© Copyright by

Cindy Xinmin Chen

2001

The dissertation of Cindy Xinmin Chen is approved.

Alfonso F. Cardenas

D. Stott Parker

Daniel Valentino

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2001

To my parents

Table of Contents

1 Introduction **1**

1.1 State of the Art 2

1.1.1 Temporal Databases 2

1.1.2 Spatial Databases 4

1.1.3 Spatio-Temporal Databases 6

1.2 Proposed Approach 9

1.3 Outline of the Dissertation 12

2 SQL^T **14**

2.1 TSQL2 15

2.2 Explicit Time Queries 17

2.2.1 Schema definition in SQL^T 20

2.2.2 Temporal Selection and Join 20

2.2.3 The VALID Clause 21

2.3 Interval-Oriented Reasoning 22

2.4 Temporal Aggregates 26

2.5 Dealing with Periods 32

2.6 Summary 36

3 Implementation of SQL^T **38**

3.1 The Temporal Query Language 38

3.1.1 Schema Definition 39

3.1.2 Temporal Selection and Join 40
3.1.3 Interval-Oriented Reasoning and Temporal Aggregates . . 42
3.2 Internal Model 49
3.2.1 Usefulness-Based Management 51
3.3 Built-in Translation From External Relations to Internal Model . 53
3.4 Summary 61
4 Properties of Spatial Objects 62
4.1 Spatial Relationships and Operations 62
4.2 Triangulation 65
4.2.1 Algorithm of Polygon Triangulation 68
4.3 Spatial Relationships Between Triangles 71
4.4 Spatial Operations on Points, Lines and Triangles 74
4.5 Spatial Relationship Between Polygons 78
4.6 Summary 79
5 A Concrete Model of Spatio-Temporal Data 80
5.1 SQL^{ST} 80
5.2 Implementation 86
5.2.1 Built-in Functions 86
5.2.2 AXL 87
5.2.3 Temporal Aggregates 91
5.2.4 Spatial Aggregates 92
5.2.5 Spatio-Temporal Aggregates 96

5.3 Performance 97
5.4 More Abstract Representations 102
5.4.1 Schema Definition 106
5.4.2 Spatio-Temporal Queries 107
5.5 Future Work 109
5.6 Summary 112

6 Conclusions 113

References 116

List of Figures

2.1 Allen's interval operators 23

4.1 An example of counterclockwise directed triangle 67

4.2 A trapezoidation of 4 line segments 69

4.3 (a) A polygon; (b) Trapezoids inside the polygon; (c) Introducing diagonals 70

4.4 Example of Relationships Between Triangles 72

5.1 Performance result of Query 48 99

5.2 Performance result of Query 49 100

5.3 Performance result of Query 50 101

5.4 Performance result of Query 51 102

5.5 Graphs representing spatio-temporal data 103

List of Tables

3.1 Semantics of Allen's operators on PERIOD 47

3.2 Semantics of Allen's operators on PERIODSET 48

5.1 Abstract model of the spatio-temporal data shown in Figure 5.5 . 105

5.2 Concrete model of the spatio-temporal data shown in Figure 5.5 . 105

Acknowledgments

I would like to express my sincere thanks and appreciation to my advisor, Professor Carlo Zaniolo, for his continuous advice, guidance, and support. His profound knowledge and kindness will always be an inspiration.

I am also grateful to Professors Alfonso F. Cardenas, D. Stott Parker and Daniel Valentino for their participation in my doctoral committee and taking the time to guide me through my dissertation.

Last, I am thankful to my colleagues Jiejun Kong for his contribution to the implementation of SQL^T and to Haixun Wang for his help with the AXL system.

Abstract of the Dissertation

Data Models and Query Languages of Spatio-Temporal Information

by

Cindy Xinmin Chen

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2001

Professor Carlo Zaniolo, Chair

In this dissertation, we extend database models and query languages to support spatio-temporal information, including representations for changing positions and shapes. Furthermore, we propose techniques to support spatio-temporal extensions on Object-Relational systems and achieve end-user extensibility.

We begin from temporal data models and query languages. Our approach is based upon a point-based representation of time enhanced with user-defined aggregates that support interval-oriented operators such as DURATION and DURING. This approach provides several advantages, since it solves the coalescing problem, minimizes extensions required from SQL, and it is applicable to all query languages.

Then, we extend the time model at the physical level to spatio-temporal databases. As the basis of our implementation, we introduce counterclockwise-directed triangles as our core spatial abstract data type. Then, user-defined-aggregates are introduced to support spatial operators such as CONTAIN and OVERLAP, and spatial-temporal operators such as MOVING_DISTANCE. Finally, we represent moving objects as sequences of snapshots of spatial objects such as

points, lines and polygons.

To reconcile logical and physical requirements, we follow an implementation approach based on a layered architecture. Thus, for temporal information, point-based representations are mapped to an interval-oriented representation at the internal level. Likewise, we map polygon-based representations into a triangle-based internal representation. We implement these mappings through query transformations, and user-defined aggregates.

A final advantage of our approach is that it allows end-users to further extend and customize their system: in fact in our SQLST system, users can extend database functionality by writing new user-defined aggregates in SQLST itself.

In summary, the spatio-temporal data models and query languages introduced in this dissertation offer several conceptual advantages; furthermore, our implementation approach provides considerable practical benefits, including compatibility with Object-Relational systems, flexibility, robustness, and ease of customization by end-users.

CHAPTER 1

Introduction

Time and space are important aspects of all real-world phenomena. Database applications must capture the time and space varying nature of the phenomena they model. For example, a doctor's office needs to maintain a record of the prescription history of patients. This information certainly varies over time: drugs are taken during a particular time; a patient may take several drugs at the same time; and a drug may be taken several times by a patient. Before prescribing drugs to a patient, the doctor needs to know the patient's history of drug usage and the other drugs that the patient is currently taking. As another example, consider a cyclone roaring through a region in the ocean. It is important to know when the cyclone will come across an island, how long a cyclone will stay on an island, which island region will be affected, and related information of a spatio-temporal nature.

In conventional databases, attributes containing temporal or spatial information are manipulated solely by the application programs, with little help from the database management system. A spatio-temporal database is one that supports aspects of both time and space. It offers spatial and temporal data types in its data model and query language.

Applications that rely on spatio-temporal databases can be found, for example, in geographical information systems, autonomous navigation, tracking, and medical imaging.

1.1 State of the Art

1.1.1 Temporal Databases

In the context of temporal databases, two time dimensions are of general interest: the *valid time* dimension and the *transaction time* dimension.

Valid time concerns the time when a fact is true in reality. The valid time of a fact is the time at which the fact occurred in the real world, independent of the recording of that fact in some databases. Valid time can be in the past, present, or in the future. While all facts have a valid time, this may not necessarily be recorded in the database, for any number of reasons.

Transaction time concerns the time when the fact is present in the database as stored data. The transaction time of a fact is that of the transaction that inserted the fact into the database, and of the transaction that removed this fact from the database.

Moreover, valid time captures the time-varying states of the reality, while the transaction time captures the time-varying states of the database. These two dimensions are orthogonal.

With many applications requiring support for temporal databases, extensive research has focused on temporal queries and reasoning [87]. A critical issue in all these approaches is the choice of model used to represent valid time: for instance, in many approaches temporal intervals are used to represent valid time states, while point-based models view the database as a sequence of snapshots.

In the interval-based time model proposed in [54], an 1-dimensional space is a nonempty finite set D, totally ordered with respect to '$<$'. The elements in D are called simply points. An interval, $I(D)$, is a set of simply points over a 1-d space D, denoted by a start point and an end point. In relations based

on interval-based time model, a temporal attribute is maintained to record time intervals. When a projection is taken, coalescing of time intervals is required. Coalescing consists of two typical steps: (1) $unfolding$ a relation with respect to the temporal attribute by replacing each tuple with a set of tuples containing only simply time points; (2) $folding$ the immediate relation with respect to the temporal attribute by repeatedly taking pairs of tuples which are identical in all attributes other than the temporal attribute and merge them, until no such pairs of mergeable tuples exist.

In point-based time model [90, 92, 7], the time domain is viewed as a discrete, countably infinite, linearly ordered set without endpoints (like the integers). The individual elements of the set represent the actual time instants, while the linear order represents the progression of time. The actual granularity of time can be chosen upon the application domain. The relationships between the time instants and the non-temporal database facts are captured by a finite set of temporal relations stored in the database.

The major drawback of the interval-based data model is the need for coalescing time intervals when a projection is taken, whereas point-based data models are free from this problem. On the other hand, many temporal queries can be expressed naturally using intervals, e.g., using Allen's interval operators. Furthermore, intervals provide a reasonably efficient representation for physical storage, while point-based conceptual models must be mapped into different representations for storage efficiency.

The language TSQL2 [82] introduced a consensus extension of SQL-92. It supports a Bi-temporal Conceptual Data Model [45], which handles both valid time and transaction time. In TSQL2, there is no explicit time column in the relations, and valid time cannot be referred in the query as a tuple attribute;

therefore, we say that *TSQL2 has an implicit-time data model*. By keeping time implicit, TSQL2 eliminates the need for a user to specify the coalescing of time periods [6]. However, implicit time makes the semantics of TSQL2 more obscure and difficult to formalize. Furthermore, the fact that one cannot work directly with time makes certain queries hard to express and requires the introduction of special temporal constructs to achieve the same goal indirectly. This brings us to the second problem, i.e., special TSQL2 constructs are designed for SQL only, and do not generalize into a universal temporal data model and query language that can be used for, say, QBE and Datalog. We refer to this problem as the *lack of universality* of TSQL2. Also, TSQL2's compatibility with Object-Relational systems was not investigated. These problems make TQSL2 not included in SQL-99.

1.1.2 Spatial Databases

The field of spatial databases has been an active area of research for over two decades. There are two common models of spatial information: field-based and object-based. The field-based model treats spatial information such as altitude, rainfall and temperature as a collection of spatial functions transforming a space-partition to an attribute domain. The object-based model treats the information space as if it is populated by discrete, identifiable, spatially-referenced entities. An implementation of a spatial data model in the context of Object-Relational databases consists of a set of spatial data types and the operations on those types.

Much work has been done on the design of spatial Abstract Data Types (ADTs) and their embedding in a query language, such as Spatial SQL [25]. Spatial SQL has two parts: a query language to describe what information to retrieve and a presentation language to specify how to display query results. Users

can issue standard SQL queries to retrieve non-spatial data based on non-spatial constraints. Moreover, they can issue Spatial SQL commands to inquire about situations involving spatial data and give instructions in the Graphical Presentation Language (GPL) to manipulate or examine the graphical presentation. The features of Spatial SQL have an Object-Oriented flavor, such as the complex abstract data type *spatial* and its subtypes for different spatial dimensions.

Paradise [65] provides what can be loosely interpreted as an Object-Relational data model. In addition to the standard attribute types such as integers, floats, strings and time, Paradise also provides a set of spatial data types including points, polygons, polylines, swiss-cheese polygons, and circles. The spatial data types provide a rich set of spatial operators that can be accessed from extended version of SQL.

Commercial examples of spatial database management include IBM DB2's Spatial Extender [42], Informix's Spatial DataBlade Module [44], Oracle's Universal Server with Spatial Option/Cartridge [61] and ESRI's Spatial Data Engine [43]. The functionalities provided by these systems include a set of spatial data types such as points, line-segments and polygons and a set of spatial operations such as inside, intersection and distance. The spatial types and operations may be made a part of a query language such as SQL, which allows spatial querying when combined with an Object-Relational database management system. The performance enhancement provided by these systems includes a multi-dimensional spatial index and algorithms for spatial access methods, spatial range queries and spatial joins.

1.1.3 Spatio-Temporal Databases

Spatio-temporal data models and query languages are a topic of growing interest. A spatio-temporal database is a database that embodies spatial, temporal, and spatio-temporal database concepts and captures simultaneously spatial and temporal aspects of data. It deals with geometries that change with time.

Constraint databases [47] can be used as a common language layer that makes the interoperability of different temporal, spatial and spatio-temporal databases possible. Constraint databases generalize the classical relational model of data by introducing generalized tuples: quantifier-free formulas in an appropriate constraint theory. For example, the formula $1950 \leq t \leq 1970$ describes the interval between 1950 and 1970; and the formula $0 \leq x \leq 2 \wedge 0 \leq y \leq 2$ describes the square area with corners $(0,0)$, $(0,2)$, $(2,2)$ and $(2,0)$. The constraint technology makes it possible to finitely represent infinite sets of points, which are common in temporal and spatial database applications.

The issue of application-independent interoperability of spatio-temporal databases is addressed in [18]. The referenced approach first introduced the TQuel data model for temporal databases and the 2-spaghetti model [55] for spatial databases, and then developed a parametric 2-spaghetti data model for spatio-temporal data. The 2-spaghetti data model [55] provides a general relational representation for geometric objects. It represents spatial objects that are composed of finite unions of closed convex polygons. Polygons are triangulated and can be represented using first-normal form relations with a fixed number of attributes. Each triangle is represented by its three corners. If the endpoints of the 2-spaghetti are functions of time, then it is called a parametric 2-spaghetti. In the TQuel data model, each relation contains two special attributes called $From$ and To to represent valid time. The value of these temporal attributes are integers or the special constants

$-\infty$ or $+\infty$. The $From$ and To values represent the endpoints of an interval; intervals in different tuples with identical non-temporal components have to be disjoint.

An example of the parametric 2-spaghetti data model represented by the constraint database is as follows.

id	x	y	x'	y'	x"	y"	From	To
r1	3	3	10	10	3	11-t	$-\infty$	$+\infty$

$object(id, x, y, t) :\!-\, id = r1, 1 \leq y, y \leq 10, 3 \leq x, x \leq 10, y \geq x + 8 - t$

The approach presented in [18] was extended in [11] by using a n-dimensional parametric rectangles (or boxes). A n-dimensional rectangle is the cross product of n intervals, each in a different dimension. The lower and upper bounds of the intervals are functions of time.

The DEDALE system [33, 34] also used linear constraints. It introduces space and time as natural components of 3-d point-sets. A polygon in the plane is seen as the infinite set of points inside its frontier. DEDALE also extends the relational algebra to contain a tuple constructor that builds tuples from atomic objects (constants and rational numbers).

In [28, 37], a framework of abstract data types for moving objects was defined. The framework takes as its outset a set of basic types that along with standard data types, such as integer and boolean; includes spatial data types, such as points and regions, and temporal data types, such as time instants. Then, it introduces type constructors that can be applied to the basic types to create new types. For

example, given an argument of type α, the function "moving" constructs a type whose values are functions of time into the domain of α. This leads to types such as *mpoint* (moving point) and *mregion* (moving region). These abstract types may be used as column types in conventional relational DBMSs, or they may be integrated in Object-Oriented or Object-Relational DBMSs.

In [99], a unified model is presented for information that uses two spatial dimensions and two temporal dimensions (valid time and transaction time). The temporal objects in this 4-dimensional space are called *bitemporal elements*. A bitemporal element is the union of a finite set of Cartesian products of intervals of valid time and transaction time. Spatial objects, assumed to be embedded in Euclidean 2-d space, are represented as simplicial complexes [98]. A *simplex* is either a single point, a finite straight line segment, or a triangular area. A simplicial complex is a collection of non-overlapping simplexes, such that if a simplex belongs to the complex then so do all its component simplexes. A simplicial complex is uniquely determined by its maximal component simplexes.

A spatio-temporal object is a unified object which has both spatial and bitemporal extents. An $ST - simplex$ is an elemental spatial object with a bitemporal reference attached. An $ST - complex$ is a collection of ST-simplexes, subject to some constraints. Firstly, the spatial projections of the constituent ST-simplex must all be distinct. Secondly, the spatial projections of the constituent ST-

simplexes must themselves form a spatial simplicial complex. Thirdly, any face of a spatial simplex occurring as a component in the ST-complex must have at least as much temporal referencing as its parent. Spatio-temporal operations such as projection and selections are also discussed in [99].

1.2 Proposed Approach

The topic of spatio-temporal databases has been the focus of research work and many solutions have been proposed in all major database conferences and specialized workshops and symposiums. This reflects the importance and significance of spatio-temporal database applications and the complexity of technical problems to be solved. The number and variety of the solutions proposed are also due to the fact that different application areas have different requirements and thus demand different solutions. However, most solutions proposed so far, particularly Database Extenders and DataBlade of commercial DBMSs [42, 44], lack flexibility and therefore generality.

Another issue is extensibility, which represents a long-sought-after but hard-to-attain goal for database systems. Relational DBMSs provided no extensibility. A relational DBMS restricts a table column to one of the certain data types including integer, floating-point number, character string, date, time, datatime, interval, numeric and decimal, and defines a precise (and hard-coded) collection

of functions and operators that are available for each data type [86]. Because the set of data types and operations is limited, many real-world problems are extremely difficult to code and, once coded, perform badly.

However, even in the latest generation of O-R (Object-Relational) systems, extensibility comes with many limitations and requires significant expertise and programming effort. Indeed, the main extensibility mechanism of O-R systems is to allow SQL queries to call external functions coded in a procedural language (such as C or Java). Function libraries for specific application domains are now marketed aggressively by vendors under different brand names, such as DataBlades, DB extenders, cartridges, or snapins. But they all share similar limitations with respect to power and flexibility, inasmuch as they can only support a predefined set of operators on single records: e.g., the functions cannot access the database tables either directly or through embedded SQL calls. Furthermore, procedural attachments to SQL, such as UDFs (User Defined Functions) or stored procedures, are notorious for being inordinately hard to develop and debug [13]. Thus current DataBlades are not conducive to end-user extensibility (even if their source code was available).

In general, while more concrete models are more suitable for efficient implementation, abstract models are better from the view point of generality and usability. The main difference between different models is the level of abstrac-

tion they support. Some models, such as the point-based time model, are more desirable from the point of usability and generality but more difficult to implement than other more storage-conscious data models such as the interval-based time model. The main concern in this scenario is to meet the practical requirement, especially the suitability, to be supported as extensions of current database systems.

We start this dissertation with a point-based representation of time enhanced with user-defined aggregates to support interval-oriented operators, followed by a different storage representation—using time intervals.

Then, we propose a concrete model for spatio-temporal data. The temporal data type remains as intervals and the spatial data type is *counterclockwise-directed triangles*. Spatio-temporal queries are expressed by user-defined aggregates, such as duration and during for temporal relationships, contain and distance for spatial ones, and moving_distance for spatio-temporal ones. We also show that users can choose an abstract model of spatio-temporal data, whose spatial data types are points, lines and polygons, and the temporal data type is time instants (points). The mapping between different models is achieved using the table expressions supported in Object-Relational systems.

Therefore, our approach minimizes the extensions required in SQL, or other relational languages, to support spatio-temporal queries. This approach provides

a better extensibility mechanism than Database Extenders or DataBlades. Also, we achieve orthogonality of temporal and spatial aspects of data, and has minimal additions to SQL. In fact, only user-defined aggregates are required to perform the spatio-temporal queries. The support of multi-layer abstraction reconciles ease of use and efficient implementation.

1.3 Outline of the Dissertation

The rest of the dissertation is organized as follows.

Chapter 2 presents a temporal data model and query language—SQL^T. SQL^T uses a point-based temporal data model and user-defined temporal aggregates to support interval reasonings. We demonstrate the universality of this approach by proposing parallel designs for SQL, QBE and Datalog. Then, in Chapter 3 we discuss an efficient implementation of SQL^T on top of DB2 Object-Relational system using user-defined functions and table expressions.

Chapter 4 presents an overview of the properties of spatial objects, and Chapter 5 presents a concrete spatio-temporal data model and query language—SQLST. Counterclockwise-directed triangles and time intervals are used at the internal level to model spatial and temporal objects. As in SQL^T, only user-defined aggregates are used to support spatio-temporal queries. Also, we discuss an abstract model of spatio-temporal information, where polygons and time in-

stants are used to model spatial objects and time. The mapping method between different models is also discussed.

Chapter 6 concludes the dissertation.

CHAPTER 2

SQL^T

Temporal reasoning and temporal query languages present difficult research problems of theoretical interest and practical importance. One issue is the chasm between point-based temporal reasoning and interval-based reasoning. Another problem is the lack of robustness and universality in many proposed solutions, whereby temporal extensions designed for one language cannot be easily applied to other query languages—e.g., extensions proposed for SQL cannot be applied to QBE or Datalog. In this chapter, we provide a simple solution to both problems by observing that all query languages support (i) single-value based reasoning and (ii) aggregate-based reasoning, and then showing that these two modalities can be naturally extended to support, respectively, point-based and interval-based temporal queries. We follow TSQL2 insofar as practical requirements are concerned, and show that its functionality can be captured by simpler constructs which can be applied uniformly to Datalog, QBE and SQL.

2.1 TSQL2

To illustrate some of the issues with TSQL2, consider a patient database with the history of prescriptions given to patients as in [103]. The schema and sample TSQL2 queries are as follows:

1. Schema definition

   Example 1 *Define the* Prescript *relation*

   ```
   CREATE TABLE Prescript
       (Name CHAR(30), Physician CHAR(30),
       Drug CHAR(30), Dosage CHAR(30),
       Frequency INTERVAL MINUTE)
   AS VALID STATE DAY
   ```

   The Prescript relation is a valid time relation. The valid time has a granularity of one day. The important observation to be made here is that valid time must be qualified via annotations, since it is not a column of the relation.

2. Temporal selection and join

   Example 2 *What drugs have been prescribed with Proventil?*

```
SELECT P1.Name, P2.Drug
FROM Prescript AS P1 P2
WHERE P1.Drug = ‘Proventil‘ AND P2.Drug <> ‘Proventil‘
   AND P1.Name = P2.Name
```

The query returns the patient's name, the drug and the maximal periods during which both that drug and Proventil were prescribed to the patient.

Undoubtedly, queries involving only selections, projections, and joins represent the best feature of TSQL2, insofar as these queries are the same as in standard SQL-92. This is accomplished by keeping the time dimension implicit, as illustrated by the fact that the valid-time column is not even mentioned in the previous query. Therefore, by default, a TSQL2 query on a valid-time relation returns a valid-time relation. Additional constructs must then be used to deviate from this behavior. For instance, the keyword SNAPSHOT must be added to produce a normal relation instead of a valid-time one. While using the keyword SNAPSHOT adds little complexity to a query, other constructs needed to override TSQL2's defaults are neither simple nor user-friendly. The VALID clause discussed next is an example.

3. VALID clause

The VALID clause is used to override the default timestamp of the resulting tuple of a query.

Example 3 *What drugs were Melanie prescribed during 1996?*

```
SELECT Drug
VALID INTERSECT(VALID(Prescript), PERIOD '[1996]' DAY)
FROM Prescript
WHERE Name = 'Melanie'
```

The query returns drugs, if any, prescribed to Melanie in 1996 and the maximal periods during which Melanie took the drugs. There will be tuples returned if some drugs were prescribed to Melanie in 1996; but, due to the VALID annotation added to the SELECT clause, only the drug history for 1996 is shown, rather than the complete history. The need for this special construct VALID is created by the fact that time in TSQL2 is kept implicit. As described in [82], VALID is only one of several new constructs required by TSQL2, which make TSQL2 hard to learn and to use.

2.2 Explicit Time Queries

The basic approach we propose here is based on a point-based temporal data model and on explicit-time queries. Our point-based temporal data model assumes:

- the use of some granularity for representing valid time—for instance we will uses days in our examples,
- every temporal relation contains an additional column, say the last column, called VTime, storing single time-granules, and
- the relation contains one row for each (time) point at which the database fact is valid.

For instance, a temporal (virtual) relation that is supported by the system, the calendar relation, is as follows:

Calendar	Year	Month	Day	VTime
	...	...	...	...
	1996	September	24	09/24/1996
	...	...	...	...

Here, we display only one tuple as a sample of our calendar relation. Also observe that we represent valid-time dates using the month/day/year notation.

This calendar relation is not a stored object, it is a virtual view that provides a user-friendly QBE interface to calendar queries, such as the following that returns all days in September 1996:

Calendar	Year	Month	Day	VTime
	P.1996	P.September		P.

In as much as this calendar query is implemented by an internal calendar function, it exemplifies the main idea of our approach: select a data model that simplifies the expression of complex temporal queries, and rely on mapping to

efficient internal representations for implementation. The idea of different representations at different levels has been long popularized by the ANSI/X3/SPARC architecture for DBMS [10]. Effective representations at the storage level are discussed in next chapter. Also at the end-user level, simple solutions exist to avoid the redundant printouts generated by point-based representations. For instance, rather than having the previous query generate 30 tuples, identical in the year and month columns, we can use the following display to present the results:

Result	Year	Month	VTime
	1996	September	09/01/1996
	...	...	...
	1996	September	09/30/1996

Since people are quite adept at filling-in the dots, this is a concise and unambiguous representation for much larger sets. The same sets could also be represented by a single tuple as follows:

```
(1996, September, 09/01/1996 ...... 09/30/1996)
```

This can either be viewed as an unnormalized tuple, where September 1996 is associated with the set of days 09/01/1996 ... 09/30/1996, or as an interval-based representation of the same information. An interval-based representation is acceptable at the user level, and has some properties that make it an interesting (although perhaps not the best) candidate at the storage level, as it will be discussed later.

2.2.1 Schema definition in SQL^T

We now describe SQL^T, our valid-time extension of SQL-92, where explicit time is used in schema declarations and queries.

Example 4 *Define the* Prescript *relation*

```
CREATE TABLE Prescript
        (Name CHAR(30), Physician CHAR(30),
        Drug CHAR(30), Dosage CHAR(30),
        Frequency INTERVAL MINUTE, VTime DATE)
```

Thus, the valid time has become the last column in our relation. The reserved keyword VTime must be used to denote the name of the valid-time column—a relation can have at most one of these columns. Similar conventions apply to the schemas defined in QBE, or to Datalog languages, such as *LDL++* [102, 52]. The expressions of temporal selection and join queries in SQL^T, QBE^T and Datalog^T are straightforward and are shown next.

2.2.2 Temporal Selection and Join

Example 5 *What drugs have been prescribed with Proventil?*

```
SELECT P2.Name, P2.Drug, P2.VTime
FROM Prescript AS P1 P2
```

WHERE P1.Drug = "Proventil" AND P2.Drug <> "Proventil"

AND P2.Name = P1.Name AND P2.VTime = P1.VTime

Prescript	Name	···	Drug	···	VTime
	_name		Proventil		_vtime
	P._name		P._drug		P._vtime

Conditions
_drug ¬= Proventil

$\texttt{query1}(\texttt{Name}, \texttt{Drug}, \texttt{VTime}) \leftarrow$

$\texttt{prescript}(\texttt{Name}, _, '\texttt{Proventil}', _, _, \texttt{VTime}),$

$\texttt{prescript}(\texttt{Name}, _, \texttt{Drug}, _, _, \texttt{VTime}),$

$\texttt{Drug} \sim= '\texttt{Proventil}'.$

2.2.3 The VALID Clause

TSQL2's VALID clause is no longer needed since, in SQL^T, the target time span can be explicitly controlled by conditions in the WHERE clause.

Example 6 *What drugs was Melanie prescribed during 1996?*

SELECT P.Drug, P.VTime

FROM Prescript AS P, Calendar AS C

WHERE P.Name = "Melanie" AND C.Year = 1996

AND P.VTime = C.VTime

The same query can be expressed as follows in QBE^T and Datalog^T:

Calendar	Year	Month	Day	VTime
	1996			_vtime

Prescript	Name	···	Drug	···	VTime
	Melanie		P._drug		P._vtime

$$\texttt{query2(Drug, VTime)} \leftarrow$$
$$\texttt{calendar(1996, _, _, VTime)},$$
$$\texttt{prescript('Melanie', _, Drug, _, _, VTime)}.$$

2.3 Interval-Oriented Reasoning

An important requirement of all temporal languages is to support Allen's interval operators such as overlap, precede, contain, equal, meet, and intersects [2] as illustrated in Figure 2.1.

Temporal languages that are based on temporal intervals [54] rely on these operators to express temporal joins. In this kind of languages, the query of Example 2 would be expressed by the condition 'P1 overlaps P2'. No explicit use of overlaps is needed in point-based semantics, since two intervals overlap if and only if they share some common points [7, 90]. This conclusion also holds for TSQL2, where equality between time points is assumed as default condition when no other temporal condition is given. In TSQL2, however, the user must

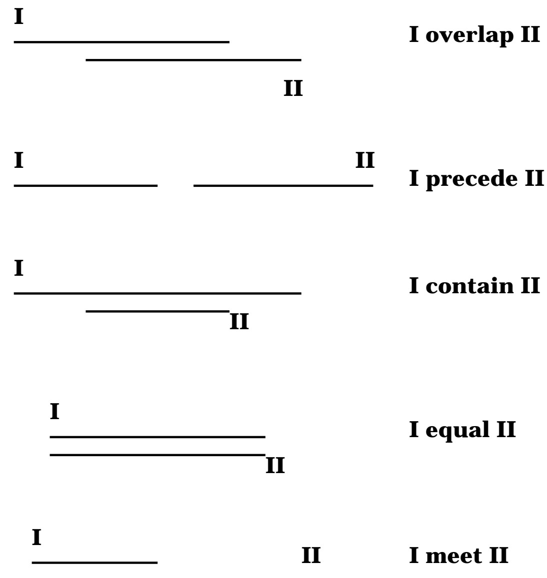

Figure 2.1: Allen's interval operators

use explicit constructs to specify the remaining Allen's operators. For instance, consider the following query:

Example 7 *TSQL2 Query: find the patients who have been prescribed Proventil, throughout 1996.*

```
SELECT SNAPSHOT Name
FROM Prescript(Name, Drug) AS P
WHERE P.Drug = "Proventil"
    AND CONTAINS(VALID(P), PERIOD "[1996]" DAY)
```

This query returns the patient's name if the patient took Proventil for the whole period of 1996. Thus VALID(P) denotes the valid time of tuples in P

represented as one or more intervals (i.e., periods in TSQL2 terminology), and PERIOD "[1996]" DAY is simply a constructor of a time period starting January 1, 1996, and ending December 31, 1996.

If a patient took Proventil during several non-contiguous time periods, then, at least one of these intervals must contain the "Year 1996" time period. This brings out the important point that TSQL2 is really dealing with sets of intervals (a *time element* in TSQL2's terminology), rather than a single interval. For now, let us ignore this point (discussed in great length in next chapter) as if every drug were only prescribed during one interval. Then, consider

```
FROM Prescript(Name, Drug) AS P
```

The attribute-list (Name, Drug) is an instance of TSQL2's "special" extension of SQL-92 called *restructuring*. The purpose of this construct is to define the attributes on which the tuples must be coalesced. Indeed Proventil might have been prescribed to the same patient by different physicians, and with different dosage and frequency. If we remove the attributes (Name, Drug) from the FROM clause of the previous query, the meaning of the query is changed into: find the patients who have been prescribed Proventil, *by the same physician, with the same dosage and frequency*, throughout 1996; this is a much stricter condition than the original one. Therefore, to support this query, we need to project out the physician, dosage, and frequency columns, and coalesce the time intervals into

maximal intervals of time during which the values of attributes (Name, Drug) remain unchanged.

Many queries that arise in the context of interval-oriented reasoning involve duration. For instance, we might want to find the names of the patients who have been prescribed some drugs for a total of more than 240 days. Again, we have to use restructuring to ensure that we accumulate the length of a prescription independent of physician, dosage and frequency (e.g., to ensure that patients do not circumvent the maximum prescription period limitations by changing physician).

Example 8 *TSQL2 Query: find the patients who have been prescribed some drug for more than 240 days.*

```
SELECT SNAPSHOT Name
FROM Prescript(Name, Drug) AS P
WHERE CAST(VALID(P) AS INTERVAL DAY) > INTERVAL 240 DAY
```

Again, VALID(P) defines one or more periods. In TSQL2, the terms period and interval denote, respectively, anchored and unanchored spans of time. Thus, in TSQL2, casting is needed to convert a set of (one or more) periods to an interval by adding up the length of each period. Therefore, the query in Example 8 adds up the lengths in days of all periods during which a patient took the same drug, and checks whether the accumulated length exceeds 240 days. A more complex

TSQL2 query is needed to find patients who took the same drug for more than 240 *consecutive* days—i.e., a continuous prescription of the same drug for more than 240 days; this is discussed in Section 2.5.

2.4 Temporal Aggregates

In a point-based temporal model, intervals are sets of contiguous points; thus *set aggregates* should be used to support interval-oriented reasoning. For instance, the last query can be formulated and expressed in SQL-92 as follows:

Example 9 *SQL-92 Query for Example 8*

```
SELECT Name
FROM Prescript
GROUP BY NAME, DRUG
HAVING COUNT(VTime) > 240
```

Observe that, unlike TSQL2 which had to introduce restructuring, no new construct is needed here. The set of attributes each period is associated with is specified explicitly and unequivocally by the group-by attributes NAME, DRUG.

While the semantics of the count aggregate faithfully expresses the concept of duration, we will introduce in SQL^T a special temporal aggregate duration to optimize:

- users' convenience of having mnemonic constructs to express the intuitive meaning of the intended operation (also we include versions that convert to different granularities, e.g., duration_month to convert to numbers of months), and
- efficiency of execution since the implementation can be optimized directly from the storage representation used for valid time (e.g., an interval-based representation).

Therefore, in our SQL^T language, the previous query will be expressed as follows:

Example 10 *SQL^T Query for Example 8*

```
SELECT Name
FROM Prescript
GROUP BY NAME, DRUG
HAVING DURATION(VTime) > 240
```

Different styles of aggregate queries are possible for SQL^T. For instance, we can express the query in Example 10 using nested sub-queries, as follows:

Example 11 *Temporal Aggregates in SQL-92 using Nested Sub-queries (same as Example 10)*

```
SELECT P.Name

FROM Prescript AS P

GROUP BY P.NAME

HAVING DURATION (

    SELECT P1.VTime

    FROM Prescript AS P1

    WHERE P1.NAME = P.NAME AND P1.DRUG = P.DRUG) > 240
```

This second form is in fact preferable, since it is more general and accommodates any number of group-by combinations—unlike the explicit group-by form used in Example 10.

Since all *query languages support aggregates*, our new temporal aggregates can be added on without perturbing the syntactic and semantic structure of the original languages. Thus, in QBE^T, our query can be expressed as follows:

Example 12 *QBE^T Query for Example 8*

Prescript	Name	···	Drug	···	VTime
	P.G._name		G._drug		_vtime

Conditions
DURATION._vtime > 240

Of particular interest, is the "G" appearing in the first and third columns of the first table; this denotes that _name and _drug serve as group-by columns for

other variables, such as _vtime, that appear in the same row without "G" [69].

For Datalog^T languages, we will use the head-aggregation syntax of *LDL++* [95]. Then, our previous query can be expressed as follows:

Example 13 *Datalog^T Query for Example 8*

```
groupdays(Name, Drug, duration⟨VTime⟩) ←
                         prescript(Name, _, Drug, _, _, VTime).

query3(Name) ←  groupdays(Name, _, TotalDays), TotalDays > 240.
```

Here an attribute name followed by the pointed brackets denotes an aggregate column. *Every aggregate column is implicitly grouped by all the non-aggregate columns in the head of the rule.* Thus, in our example, we compute the aggregate duration of VTime with respect to the two other columns Name and Drug.

In general, the limitations caused by implicit group-by attributes are easily overcome given the great flexibility of user-defined aggregates in *LDL++*, Version 5 [102, 52]. In particular, we have defined a binary aggregate called contain which is true when one set of time points contains all the time points in the other set. Thus our query of Example 7 can be formulated as follows:

Example 14 *Datalog^T Query for Example 7: find the patients who have been prescribed Proventil, throughout 1996.*

$$\texttt{query4}(\texttt{Name}, \texttt{contain}\langle(\texttt{VTime1}, \texttt{VTime2})\rangle) \leftarrow$$

$$\texttt{prescript}(\texttt{Name}, _, '\texttt{Proventil}', _, _, \texttt{VTime1}),$$

$$\texttt{calendar}(1996, _, _, \texttt{VTime2}).$$

In an interval-based representation, VTime1 and VTime2 are implemented as two sets of intervals. Then, contain⟨(VTime1, VTime2)⟩ is implemented by checking that, for each interval I_2 in VTime2, there is an interval I_1 in VTime1, such that I_1 contains I_2.

The binary aggregate contain can also be used in QBE^T quite naturally:

Example 15 *QBE^T Query for Example 7*

Prescript	Name	···	Drug	···	VTime
	P.G._name		Proventil		_vtime1

Calendar	Year	Month	Day	VTime
	1996			_vtime2

conditions
CONTAIN.(_vtime1, _vtime2)

In SQL^T, the same query is expressed most naturally using the nested subquery technique discussed previously:

Example 16 *SQL^T Query for Example 7*

```
SELECT P.Name

FROM Prescript AS P
```

```
GROUP BY P.Name
HAVING
    ((SELECT P1.VTime
    FROM Prescript AS P1
    WHERE P1.Name = P.Name AND P1.Drug = P.Drug
        AND P1.Drug = "Proventil")
    CONTAIN
    (SELECT C.VTime
    FROM Calendar AS C
    WHERE C.Year = 1996))
```

The semantics of our new temporal aggregates can be defined from existing SQL-92 aggregates in a very natural fashion. For instance, contain(S1, S2) can be defined as a shorthand of

$$\mathsf{COUNT(S1) = COUNT(S1 \cap S2)}$$

where the set intersection can be expressed using joins. Likewise, precede(S1, S2) and meet(S1, S2) can be respectively defined as max(S1) $<$ min(S2) and max(S1) $=$ min(S2).

2.5 Dealing with Periods

TSQL2's basic time element consists of a set of periods. For a drug, therefore, the duration of each prescription period is added up when computing the length of a prescription. To deal with individual prescription periods, TSQL2 introduces the special keyword PERIOD. Thus, to find drugs prescribed for more than 240 *consecutive* days, we have the following query:

Example 17 *TSQL2 Query: find the patients who have been prescribed some drugs for more than 240 consecutive days.*

```
SELECT SNAPSHOT Name
FROM Prescript(Name, Drug) (PERIOD) AS P
WHERE CAST(VALID(P) AS INTERVAL DAY) > INTERVAL 240 DAY
```

This solution suffers from several problems including the fact that (i) partitioning violates the TSQL2's data model [82] and (ii) we do not know how to extend this construct to query languages where there is no FROM clause.

Again, TSQL2's problem can be solved using a new aggregate called period which basically enumerates the periods in ascending temporal order. Time-points that fall within the same *consecutive* period of time are given the same period number (PerNo), and a different number is used for each period.

We can now define the following view:

Example 18 *A View Enumerating Periods*

```
CREATE VIEW PartitionedP(Name, Drug, PerNo, VTime)
    AS SELECT P1.Name, P1.Drug, PERIOD(P2.VTime), P1.VTime
    FROM Prescript AS P1 P2
    WHERE P1.Name = P2.Name AND P1.Drug = P2.Drug
    GROUP BY P1.Name, P1.Drug, P1.VTime
```

Now, PerNo can be used as one of the group-by attributes:

Example 19 *SQL^T Query for Example 17*

```
SELECT Name
FROM PartitionedP
GROUP BY Name, Drug, PerNo
HAVING DURATION(VTime) > 240
```

An important advantage of this approach is that SQL^T can be defined completely using SQL-92. To compute PerNo for any time-point in an interval we must count the number of start-points of periods before it. Thus, the view PartitionedP could also have been defined as follows:

Example 20 *The meaning of* **PERIOD** *in SQL-92*

```
CREATE VIEW PartitionedP(Name, Drug, PerNo, VTime)
```

```
AS SELECT P1.Name, P1.Drug, COUNT(P2.VTime), P1.VTime
FROM Prescript AS P1 P2
WHERE P1.Name = P2.Name AND P1.rug = P2.Drug
    AND P1.VTime > = P2.VTime AND NOT EXIST
        (SELECT P3.*
        FROM Prescript AS P3
        WHERE P3.VTime = P2.VTime − 1 AND P3.Name = P2.Name
            AND P3.Drug = P2.Drug)
GROUP BY P1.Name, P1.Drug, P1.VTime
```

This definition is primarily of theoretical interest. The direct implementation of this aggregate is much more efficient—actually very efficient as we assume that the intervals are stored in ascending temporal order.

The previously created view can be useful for other queries as well. For instance, a query to find drugs whose first period of prescription to a patient was totally contained in 1996, is simply expressed as follows:

Example 21 *SQL^T Query: find drugs whose first prescription period is contained in 1996*

```
SELECT P.Drug
FROM PartitionedP AS P
```

```
WHERE P.PerNo = 1

GROUP BY P.Drug

HAVING

    ((SELECT C.VTime

    FROM Calendar AS C

    WHERE C.Year = 1996)

    CONTAIN

    (SELECT P1.VTime

    FROM Prescript AS P1

    WHERE P1.Name = P.Name AND P1.Drug = P.Drug))
```

This query is hard to express in TSQL2 but it is easy to be expressed in QBE^T and Datalog^T extended with a period aggregate and a contain aggregate.

Example 22 *QBE^T Query for Example 21*

Prescript	Name	...	Drug	...	VTime
	G._name		G._drug		PERIOD.(_vtime)

PartitionedP	Name	Drug	PerNo	VTime
	_name	_drug	_perno	_vtime

PartitionedP	Name	Drug	PerNo	VTime
		P.G._drug1	_perno1	_vtime1

Calendar	Year	Month	Day	VTime
	1996			_vtime2

Conditions
_perno1 = 1 AND CONTAIN(_vtime2, _vtime1)

Example 23 *Datalog^T Query for Example 21*

```
partitionedP(Name, Drug, period⟨VTime⟩) ←
                        prescript(Name, _, Drug, _, _, VTime),

query5(Drug, contain⟨(VTime2, VTime1)⟩) ←
                        partitionedP(_, Drug, 1, VTime1),
                        calendar(1996, _, _, VTime2).
```

In Datalog^T, period⟨VTime⟩ must return the original argument VTime along with its period number PerNo. On the other hand, contain evaluates to either true or false and returns zero arguments. Therefore, only the Name, Drug values for which the aggregate contain⟨(VTime2, VTime1)⟩ evaluates to true are produced in the head of the rule. The flexibility of having zero, one, or several values returned is supported in Version 5.1 of *LDL*++ [102, 52].

2.6 Summary

In this chapter, we have taken a minimalist approach using a point-based representation. While the standard SQL aggregates, such as sum, count, avg, min

and max, would suffice in terms of expressive power, we have added temporal aggregates to boost users' convenience and implementation efficiency.

The benefits of point-based representation were first explored by Toman [91, 92]. Here, we have improved on that work by introducing temporal aggregates that model Allen's temporal operators and TSQL2's partitioning by user-friendly constructs amenable to efficient implementation [95]. We proved the power and generality of this approach by showing that it can express all valid-time TSQL2 queries. We have also shown the *universality* of the approach (i.e., its validity with different query languages).

CHAPTER 3

Implementation of SQL^T

This chapter investigates the problem of supporting SQL^T. TENORS (Temporally ENhanced O-R System) can express temporal queries as powerful as those of TSQL2 with only minimal extensions to standard SQL by using constructs such as user-defined functions and table expressions supported in Object-Relational systems. In this chapter, we describe the indexing and clustering strategies used for TENORS, and the mapping of external queries into internal queries to optimize efficient execution on a DB2 Object-Relational database. Our storage strategy uses an interval-based representation, where relations are time-segmented using a manageable *current usefulness* criterion; the execution strategy relies on user-defined functions and aggregates.

3.1 The Temporal Query Language

In the point-based model [7, 90], the database is viewed as a sequence of snapshots from user's standpoint. The main advantage of this model is that it eliminates the

need for explicit coalescing after projection that is instead required by interval-based data model. In other words, the point-based model lays the burden of coalescing on database rather than on users, thus becomes more user-friendly. It assumes that,

- Some granularity is used for representing valid time. Basically, atomic granules, also called chronons, can be defined using the standard SQL-92 data types of DATE, TIMESTAMP. In this chapter, DATE is used in all examples.
- Every temporal relation contains an additional column, say the last column, called VTime, storing chronons.
- The relation (virtually) contains one row for each chronon at which the database fact is valid.

3.1.1 Schema Definition

Temporal relations in TENORS contain an explicit valid time column that is part of the schema declarations and is then used in queries.

Example 24 *Define the* Employ *relation and* Position *relation.*

```
CREATE TABLE Employ
        (Name CHAR(30), Title CHAR(30), Title_Level DECIMAL(2),
```

```
        Misc CHAR(256), VTime DATE,
        PRIMARY KEY(Name,Title,Title_Level));
CREATE TABLE Position
        (Title CHAR(30), Title_Level DECIMAL(2), Salary DECIMAL(10,2),
        VTime DATE,
        PRIMARY KEY(Title,Title_Level));
```

Thus, the valid time has become the last column in the relations. The reserved keyword VTime must be used to denote the name of the valid-time column—a relation can have at most one of such column. The expression of temporal selection and join queries in TENORS is straightforward and shown next.

3.1.2 Temporal Selection and Join

In point-based model, simple conditions can be applied to the temporal column VTime to select a snapshot of the relation or the history of the qualified tuples during a certain period of time. For example, to find the positions Melanie was holding on certain dates, only the time span on VTime is needed to be adjusted:

Example 25 *What title was Melanie holding on June 1, 1996?*

```
SELECT Title
FROM Employ
```

```
WHERE Name = “Melanie” AND VTime = “6/1/1996”
```

Likewise, if users want to see Melanie's employment history during 1996, in TSQL2 users have to use the special construct VALID which is required for that purpose. In TENORS, users can simply replace the last qualification of Example 25 with

```
VTime >= “1/1/1996” AND VTime < “1/1/1997”
```

Moreover, temporal joins are handled just as regular natural joins. By assuming a virtual point-based model, TENORS borrows the join semantics directly from SQL without introducing any extension.

Example 26 *Show the salary history for directors in the 1990s.*

```
SELECT Employ.Name, Position.Salary, Employ.VTime
FROM Employ, Position
WHERE Employ.Title = “Director” AND Employ.Title = Position.Title
    AND Employ.VTime = Position.VTime AND Employ.VTime >= “1/1/1990”
    AND Employ.VTime < “1/1/2000”
```

Temporal joins involve equality of points in time. In TENORS, the notion of “*same time*” is naturally captured by the equality

```
Employ.VTime = Position.VTime
```

$$Employ.VTime = Position.VTime$$

In TSQL2, this equality is omitted from the query since it is implicitly assumed by the system for every query. This behavior is different from SQL where any join condition must be *explicitly* specified to meet varying circumstances. Therefore, TSQL2's "*same time*" implicit assumption simplifies join queries as that above, but then requires additional constructs such as restructuring, coupling and VALID for more complex queries.

The main advantage offered by both TSQL2 and TENORS is that they are both free of the coalescing problem that besets languages for interval-based representations.

3.1.3 Interval-Oriented Reasoning and Temporal Aggregates

Temporal intervals, which in TSQL2 are called *periods*, provide a simple condensed representation of a set of contiguous time granules, which has been widely used in temporal databases.

In fact, for several non-overlapping intervals associated with a tuple, two different representations are possible: either they can be treated as a set thus constitute a single tuple, or each interval is associated with the tuple separately thus constitute multiple tuples. Previous work has used both representations based on set of intervals (called the *time element* in TSQL2) [82], and representation based

on individual interval [54]. TENORS provides two temporal aggregates PERIODSET and PERIOD to map sets of chronons into sets of intervals and individual intervals, respectively.

This aggregate-based approach introduces minimal extension to Object-Relational model since user-defined functions and user-defined aggregates are already available in some O-R databases [84]. For instance, the query "find all the positions held by Melanie for more than 90 days" can be expressed using aggregate PERIODSET and function span:

Example 27 *Melanie's positions of more than 90 days as a whole.*

```
SELECT Title
FROM  Employ
WHERE Name = "Melanie"
GROUP BY Title
HAVING SPAN(PERIODSET(VTime)) > 90
```

where the aggregate PERIODSET delivers a set of intervals and the function span sums up the overall length for the entire time span. In this example, after selecting the records for Melanie, users need to coalesce all the intervals during which she had a given title (as users have projected out the columns Title_Level and Misc). TSQL2 instead uses a special construct called "*restructuring*" to

express what is naturally expressed in TENORS by the standard group-by clause and aggregation.

While PERIODSET is an aggregate, span is a scalar function which accepts a composite object (a set of intervals or a list of individual intervals) as argument and returns the sum of each interval's length.

Since Melanie could have held the same title during several time periods, with interruptions in between; then a user might want to find the positions held by Melanie for more than 90 *consecutive* days. The PERIOD aggregate must be used for this purpose:

Example 28 *Melanie's positions of more than 90* consecutive *days.*

```
SELECT Title
FROM  Employ
WHERE Name = "Melanie"
GROUP BY Title
HAVING SPAN(PERIOD(VTime)) > 90
```

The PERIOD aggregate seems a bit unusual since it may return multiple tuples with identical group value and different time intervals. Generalized user-defined aggregates that can return more than one answer have been proposed by several researchers [41, 95, 96]. For instance, in a time series, rather than the single

global maximum, it need to return the multiple local maxima. Another important application is online aggregates [41], where a sequence of "early returns" is returned for each group value to provide a good approximation of the final value of the average.

The two temporal aggregates, PERIOD and PERIODSET of TENORS, transform the basic representation (values, chronons) into (values, period), and (values, set-of-periods) representations, respectively. Depending on selection of granularity, each of these two aggregates come in two versions—one for the granularity of DATE and one for the granularity of TIMESTAMP according to the datetime types of SQL-92.

Scalar Functions

The new data type DATEPERIOD (or TIMESTAMPPERIOD) supports some scalar functions. In fact, DATEPERIOD consists of a pair of dates (or timestamps) and supports the unary functions start that returns the first point of the pair, and end that returns the second point[1]. Certainly it is mandatory that end > start, otherwise it is treated as a NULL value.

The use of the PERIOD aggregate followed by interval-based functions such as span and contains provides the essential mechanism for expressing the well-known operators of Allen's interval algebra [2].

[1] Semi-closed periods are used . The advantage of semi-closed period is no extra computing in coalescing when two periods are consecutive.

Consider the following query: "find those people whose incumbency contains a Joe's incumbency period on the same titles" can be expressed in terms of contains as follows:

Example 29 *People's incumbency that contains one of Joe's incumbency period on same titles.*

```
SELECT R.Name, R.Title, PERIOD(R.VTime)
FROM (SELECT Title, PERIOD(VTime)
          FROM Employ
          WHERE Name = "Joe"
          GROUP BY Title
      ) AS L(Title, Prd),
      Employ AS R
WHERE L.Title = R.Title
GROUP BY R.Name, R.Title
HAVING CONTAINS(L.Prd, PERIOD(R.VTime))
```

The minimalist approach has not introduced new TENORS constructs for Allen's operators such as overlap, precede, meet and the functions introduced above, since they can be easily expressed by the combination of existing SQL expressions without any loss of performance (Table 3.1).

Although contains could also be expressed using start and end, its explicit

Operator	*Semantics defined on* PERIODTYPE
SPAN(x)	SPAN: PERIODTYPE $\mapsto$ INTERVAL $end(x) - start(x)$
x OVERLAPS y	OVERLAPS: PERIODTYPE $\times$ PERIODTYPE $\mapsto$ BOOLEAN $start(x) < end(y) \wedge start(y) < end(x)$
x PRECEDES y	PRECEDES: PERIODTYPE $\times$ PERIODTYPE $\mapsto$ BOOLEAN $start(x) < start(y)$
x CONTAINS y	CONTAINS: PERIODTYPE $\times$ PERIODTYPE $\mapsto$ BOOLEAN $start(x) \leq start(y) \wedge end(x) \geq end(y)$
x MEETS y	MEETS: PERIODTYPE $\times$ PERIODTYPE $\mapsto$ BOOLEAN $start(x) = end(y) \vee end(x) = start(y)$
x EQUALS y	EQUALS: PERIODTYPE $\times$ PERIODTYPE $\mapsto$ BOOLEAN $start(x) = start(y) \wedge end(x) = end(y)$
x INTERSECTS y	INTERSECTS: PERIODTYPE $\times$ PERIODTYPE $\mapsto$ PERIODTYPE x OVERLAPS $y \models [max(start(x), start(y)),$ $min(end(x), end(y)))$

Table 3.1: Semantics of Allen's operators on PERIOD

inclusion allows significant opportunities for query optimization, which will be discussed in Section 3.3.

Similar considerations also apply to the PERIODSET data types (Table 3.2), which contain an identically-named function for each one of PERIOD. This function overloading is quite natural, since the application of, say, the span function to a period T yields the same result as the application of its dual span to the singleton set containing only T. Similar considerations hold for all functions.

As illustrated in Example 29 the use of table expression provides a very powerful mechanism for putting TENORS' temporal aggregates to work. For instance, the query in Example 28 can also be expressed using this useful syntactic device:

Operator	*Semantics defined on* PERIODSETTYPE
SPAN(X)	SPAN: PERIODSETTYPE $\mapsto$ INTERVAL $\Sigma_{\{x \in X\}}$SPAN(x)
X OVERLAPS Y	OVERLAPS: PERIODSETTYPE $\times$ PERIODSETTYPE $\mapsto$ BOOLEAN $\exists x \in X, \exists y \in Y, x$ OVERLAPS y
X PRECEDES Y	PRECEDES: PERIODSETTYPE $\times$ PERIODSETTYPE $\mapsto$ BOOLEAN $\forall x \in X, \forall y \in Y, x$ PRECEDES y
X CONTAINS Y	CONTAINS: PERIODSETTYPE $\times$ PERIODSETTYPE $\mapsto$ BOOLEAN $\forall y \in Y, \exists x \in X, x$ CONTAINS y
X MEETS Y	MEETS: PERIODSETTYPE $\times$ PERIODSETTYPE $\mapsto$ BOOLEAN $\exists x \in X, \exists y \in Y, x$ MEETS y
X EQUALS Y	EQUALS: PERIODSETTYPE $\times$ PERIODSETTYPE $\mapsto$ BOOLEAN $(\forall y \in Y, \exists x \in X, x$ EQUALS $y) \wedge (\forall x \in X, \exists y \in Y,$ y EQUALS $x)$
X INTERSECTS Y	INTERSECTS: PERIODSETTYPE $\times$ PERIODSETTYPE $\mapsto$ PERIODSETTYPE $\bigcup\{x$ INTERSECTS $y \mid \forall x \in X, \forall y \in Y\}$

Table 3.2: Semantics of Allen's operators on PERIODSET

Example 30 *Restructuring through TABLE Expression*

```
SELECT Title
FROM (SELECT Title, PERIOD(VTime)
          FROM Employ
          WHERE Name = “Melanie”
          GROUP BY Title
      ) AS T(Title, Prd)
WHERE SPAN(T.Prd) > 90
```

TSQL2 would use the restructuring and partition construct, instead of the ta-

ble expression. Interesting enough, both TENORS' table expression and TSQL2 restructuring are used in the FROM clause, suggesting that (i) there can be a simple mapping from TSQL2 to TENORS, and (ii) this new SQL construct might be used to reduce the number of new constructs needed in TSQL2. In general, many of the techniques presented in this chapter can be adapted to support much of TSQL2 on an O-R system.

3.2 Internal Model

The internal representation TIM (for Temporal Internal Model) was selected to (i) efficiently support snapshot queries, intersection queries, and temporal join; (ii) simplify the mapping into current O-R systems for both queries and indexing. Practical considerations also restricted the choices to approaches which could be implemented on top of B^+-trees, since this is the only indexing structure supported by all commercial systems and proved to be a de facto standard.

There has been a substantial amount of excellent work on valid-time indexing. However, they could not be directly used because of a number of reasons. For example, some representation were designed for such as transaction time databases or temporal events and are not directly applicable to valid-time databases [53, 75]. Other representations, including [56, 23, 26, 87], are for valid-time databases, but cannot be supported easily on top of B^+-trees. Finally, [60, 31] proposed tempo-

ral indexing schemes based on standard B^+-trees. However, in their proposals, various indices must be built on the same attribute and query optimizer must know how to choose the right one under various circumstances—this feature is not yet supported by current O-R systems. Also none of the above proposals considered clustering of temporal data via existing indexing strategies.

On the contrary, TENORS adopts a $usefulness$ based scheme that builds on standard B^+-trees, and introduces no change in storage modules of existing DBMSs. Hence it can easily support primary and secondary indices to exploit temporal and non-temporal qualification and a combination of the two.

The most natural application for temporal databases occurs in the evolutionary scenario, where an enterprise that currently stores and queries conventional time-varying relations (such as employees and customers) is upgraded to store and query the history of such relations. Then, for each query "find objects satisfying a qualification Q" there will be natural counterparts such as "find the objects satisfying Q" at a given time t (snapshot query), at a given time interval $[t, t']$ (intersection query), and along complete history (history query). This scenario also suggests that (i) performance of queries on a conventional databases provides the natural metrics for evaluating the performance of their temporal versions; (ii) a suitable selection of indices on conventional databases can be used to automatically generate a suitable selection of indices for the history database.

3.2.1 Usefulness-Based Management

In TIM, every tuple stores an interval of validity for the non-temporal information by means of two additional attributes, VTstart and VTend, to represent the set of VTime chronons that are contained in the semi-closed period [VTstart, VTend). Also the history of a relation is partitioned into k consecutive temporal segments, $S_1, S_2, \ldots, S_k$.

A temporal relation is segmented as follows. Initially it is at S_1 and all tuples are valid at the moment, thus it defines the initial *usefulness* as 100%. As time goes by, some tuples become invalid and usefulness at moment t is defined to be

$$U(t) = \frac{V(t)}{S(t)}$$

where $V(t)$ is the count of all tuples still valid at t, and $S(t)$ is the count of all tuples in current segment. TENORS assumes that users have specified a minimum usefulness level U_{min}, then current segment is split at the moment when usefulness falls below that level (i.e., $U < U_{min}$).

The split is achieved by *null change*: (i) a new segment is allocated by increasing current segment number by one; (ii) all currently invalid tuples are untouched thus left in previous segments; (iii) all currently valid tuples are broken into two pieces—the invalid piece is left in the previous segment and the valid piece is copied into the new segment.

Therefore, whenever a temporal segment is instantiated, all tuples are valid

Algorithm 1 Null Change Algorithm

Require: Initially, $Tseq = 1, U = 100\% > U_{min}$.

1: **for** Each insertion, deletion, and update of active tuples **do**
2: $\quad N_{active}$ = count(active tuples in current slice)
3: $\quad N_{all}$ = count(all tuples in current slice)
4: $\quad U = \frac{N_{active}}{N_{all}}$
5: $\quad$ **if** current relation slice is not empty $\wedge\ U < U_{min}$ **then**
6: $\quad\quad Tseq' = Tseq + 1$
7: $\quad\quad$ Insert all active tuples in slice $Tseq'$
8: $\quad\quad$ Update VTend of all active tuples in $Tseq$ as R.T. (special mark)
9: $\quad\quad Tseq = Tseq'$
10: $\quad$ **end if**
11: **end for**

and U is 100%. U decreases until it falls below U_{min}, then a null change would create a new segment.

This usefulness-based scheme naturally clusters tuples according to their VTime values, thus confers vast opportunities to speed up snapshot queries and intersection queries. The price is that tuples with long VTime values are duplicated thus cause certain redundancy. Fortunately, the tradeoff between storage redundancy and retrieval speedup is user-controllable:

- If $U_{min} = 0$, then null change never happens and a temporal relation is left unsegmented. Let us denote the size of this unsegmented relation as N_0.
- For $0 < U_{min} < 1$, the usefulness in each segment of R equals U_{min} except the current one, which is somewhere between U_{min} and 100%. For statistical reasons, we may ignore the exception and assume each segment is at the level of U_{min}. If there are n_i tuples in a temporal segment S_i, then the

number of invalid tuples in the segment is $(1 - U_{min}) \times n_i$, and those tuples are the only tuples that contribute to a coalesced relation. Therefore, if $N_{U_{min}}$ denotes the size of a segmented relation, then $N_0 = (1 - U_{min}) \times N_{U_{min}}$ and $N_{U_{min}} = \frac{N_0}{1-U_{min}}$.

For example, when $U_{min} = 0.6$ or $U_{min} = 0.5$, the size of segmented relation obtained is respectively 2.5 or 2 times the size of its unsegmented dual.

In the above discussion, only inserting valid tuples into temporal relations is considered. It is not difficult to handle retroactive insertion in the usefulness-based scheme—an obsolete tuple will be inserted into corresponding temporal segments with necessary splits. This makes the usefulness level of those segments higher than U_{min} but it is acceptable to snapshot and intersection queries. On the other hand, a tuple valid at future time will always stay in current segment because null changes can not sever an invalid piece from it.

3.3 Built-in Translation From External Relations to Internal Model

The temporal segmentation just described is incorporated in TIM's temporal indexing as follows:

- The VTime column in the original relation is replaced by the three columns: Tseq, VTstart, VTend where Tseq is the relation-segment number and VTstart, VTend denote the beginning and end of the interval.

- TENORS also allows the user or database administrator to declare primary (aka., clustered) and secondary (i.e., non-clustered) indices on the external relation—on any combination of attributes *except* VTime. Then these indices are automatically translated into indices on TIM relations as follows:

  1. If K is the primary index on the external relation, then (Tseq,K,VTstart) will be the primary index on the TIM relation.

  2. If K is a secondary index on the external relation, then (K,Tseq,VTstart) will be a secondary index on the TIM relation.

- For each external relation R, an auxiliary internal relation R_{Tseq} is created to record start time and end time of every segment of R.

For instance, Example 24 will be translated to

Example 31 *Define the* Employ *relation and* Position *relation*

```
CREATE TABLE Employ
        (Tseq INTEGER, Name CHAR(30),
        Title CHAR(30), Title_Level DECIMAL(2),
        Misc CHAR(256), VTstart DATE, VTend DATE);
```

```
CREATE TABLE Employ_Tseq
        (Tseq INTEGER, VTstart DATE, VTend DATE);


CREATE TABLE Position
        (Tseq INTEGER, Title CHAR(30), Title_Level DECIMAL(2),
        Salary DECIMAL(10,2), VTstart DATE, VTend DATE);
CREATE TABLE Position_Tseq
        (Tseq INTEGER, VTstart DATE, VTend DATE);
```

And TENORS index management statements:

```
CREATE INDEX i1 ON Employ(Name,Title,Title_Level) CLUSTER;
CREATE INDEX i2 ON Position(Title,Title_Level) CLUSTER;
CREATE INDEX i3 ON Position(Salary);
```

are automatically translated into the following internal ones:

```
CREATE INDEX i1 ON Employ(Tseq,Name,Title,Title_Level,VTstart) CLUSTER;
CREATE INDEX i2 ON Position(Tseq,Title,Title_Level,VTstart) CLUSTER;
CREATE INDEX i3 ON Position(Salary,Tseq,VTstart);
```

Besides translation of external tables and their indices into their internal counterparts, TENORS also straightforwardly translates external queries to TIM queries. For instance, the intersection query of Example 25 is translated into the following query which exploits R_{Tseq} table and primary index on Tseq to access only those relevant temporal segments rather than the whole relation.

Example 32 *What titles was Melanie holding during 1996?*

```
SELECT E.Title,
        PERIOD(max(E.VTstart, "1/1/1996"), min(E.VTend, "1/1/1997"))
FROM Employ AS E, Employ_Tseq AS ET
WHERE ET.VTstart < "1/1/1997" ET.VTend > "1/1/1996"
    AND E.Name = "Melanie" AND E.Tseq = ET.Tseq
    AND E.VTstart < "1/1/1997" AND E.VTend > "1/1/1996"
GROUP BY E.Title
```

The query finds temporal segments overlapping with user-specified time span ("1/1/1996" to "12/31/1996" in this case), then calls PERIOD aggregate and outputs coalesced result. It is important to observe that there are two different versions of temporal coalescing—one for screen display and one for physical storage. The previous one is called *external coalescing* and the later one *internal*

coalescing. External coalescing removes Tseq attribute while internal coalescing must keep Tseq attribute in the result valid-time relation for potential use in later operations.

Considering the case of internal coalescing, the result relation automatically keeps primary indexing on the input table's Tseq because data is obtained segment by segment from the input relation. Therefore, later temporal operations on the result relation are still able to exploit the primary index on Tseq.

Temporal Joins The benefits of segmentation are dramatic when computing temporal joins. For unsegmented valid-time relations temporal joins can have intractable performance, since access to right side relation is multiplied as left side relation grows in size, and vice versa. However, this is not true for segmented temporal relations.

For segmented temporal relations, a temporal segment in left relation is only joinable with overlapping temporal segments in right relations. Example 32 shows how to efficiently pick up overlapping temporal segments for any period. For temporal join, TENORS introduces a transient *mediator table* $M(L_{TSEQ}, R_{TSEQ})$ computed as

$$\Pi_{L_{TSEQ},R_{TSEQ}}(L_TSEQ(L_{TSEQ}, L_{START}, L_{END})$$

$$\bowtie' R_TSEQ(R_{TSEQ}, R_{START}, R_{END}))$$

where $\bowtie' = \bowtie_{intersect} = \bowtie_{(L_{START} < R_{END}) \wedge (R_{START} < L_{END})}$. Thus it derives overlapping Tseqs for left relation L and right relation R. Example 26 is then translated into:

Example 33 *Show the salary history for directors in the 1990s.*

```
SELECT L.Name, R.Salary,
        PERIOD(max(L.VTstart, R.VTstart, "1/1/1990"),
        min(L.VTend, R.VTend, "1/1/2000"))
FROM Employ AS L,
        (SELECT LT.Tseq, RT.Tseq
         FROM Employ_Tseq AS LT, Position_Tseq AS RT
         WHERE LT.VTstart < "1/1/2000" AND LT.VTend > "1/1/1990"
             AND LT.VTstart < RT.VTend AND LT.VTend > RT.VTstart
        ) AS M(LTseq, RTseq),
         Position AS R
WHERE L.Title = "Director" AND L.Title = R.Title
    AND M.LTseq = L.Tseq AND M.RTseq = R.Tseq
    AND L.VTstart < R.VTend AND L.VTend > R.VTstart
    AND L.VTstart < "1/1/2000" AND L.VTend > "1/1/1990"
    AND R.VTstart < "1/1/2000" AND R.VTend > "1/1/1990"
```

```
GROUP BY L.Name, R.Salary
```

Compared to Example 26, the first two qualifications are unchanged. The other qualifications implement temporal join via the mediator table and intersection.

Interval-Oriented Queries As mentioned in Section 3.1.3, an important optimization performed by TENORS is for queries involving period containment, such as the query in Example 29. In this query, users can take advantage of the fact that for an interval to be contained into the other, it must also overlap it. In fact, the occurrence of the contains construct in a query prompts the query optimizer to apply this optimization. Thus the query of Example 29 is mapped into:

Example 34 *People's incumbency that contains one of Joe's incumbency period on same titles.*

```
SELECT R.Name, R.Title, PERIOD(R.VTstart, R.VTend)
FROM (SELECT Tseq, Title, PERIOD(VTstart,VTend)
          FROM Employ
          WHERE Name = "Joe"
          GROUP BY Tseq, Title
```

```
        ) AS L(Tseq,Title,VTstart, VTend),
        Employ AS R
WHERE L.Tseq = R.Tseq AND L.Title = R.Title
    AND L.VTstart < R.VTend AND R.VTstart < L.VTend
GROUP BY R.Name, R.Title
HAVING L.VTstart >= R.VTstart AND L.VTend <= R.VTend
```

This query involves temporal join to test overlapping. An interesting observation is that since L and R share the same Tseq table, thus the transient mediator table is saved as temporal segments in the two relations that can be joined directly.

Other fundamental temporal query optimizations involved in this query include (1) dynamically applying temporal operators span and contains to periods generated by the aggregate PERIOD, (2) exploiting physical storage and indices, and (3) temporal query rewriting for Allen's operators. For instance, start and end functions are rewritten as VTstart and VTend directly. span and contains are rewritten as combination of existing SQL operators and logical expressions of VTstart and VTend.

3.4 Summary

In this chapter, we discuss an implementation of SQL^T—TENORS, which used time intervals at the physical level. TENORS has shown that O-R constructs such as user-defined functions and table expressions in the FROM clause can play a role as important as ADTs in supporting temporal extensions. At the physical level, TENORS has successfully addressed issues on data model, query language, and temporal clustering, that cannot be easily solved in an ADT-based approach. TENORS used a usefulness-based storage organization for performance, scalability, and automatic generation of temporal indices.

Future work includes designing user-defined index strategies which are now being included in several DBMSs. To the extent that user-defined index strategies will simplify the mapping from the external model to the internal one, they will further reinforce our positive conclusion on the practicality of building valid-time databases on O-R systems.

CHAPTER 4

Properties of Spatial Objects

Spatial objects possess some unique properties. There are many kinds of relationships between two or more spatial objects; and there are many types of operations that can be applied to spatial objects.

One important question in a spatial database is how to represent geometry. In this chapter, we introduce the concept of *counterclockwise directed triangle*, which is used together with points and lines to model spatial data, followed by a polygon triangulation algorithm developed by Seidel [73].

Next, we discuss the spatial relationships between triangles and spatial operations on points, lines and triangles and show how to define spatial relationships between two polygons based on the relationships between triangles.

4.1 Spatial Relationships and Operations

The fundamental properties of spatial objects can be classified in non-geometric and geometric properties [72]. Non-geometric properties describe attribute-based

data which are usually expressed by standard alpha-numerical data, e.g., the population or name of a city, the cost and the energy consumption of a house. Geometric properties can be distinguished in *metric* and *topological* features.

Metric features describe shape and location of spatial objects in a reference frame. A reference frame is the standardized spatial background into which a set of spatial objects is conceptually embedded, e.g., the Cartesian coordinate system. Each point is attached to a certain reference (coordinate) point. The shape of a spatial object describes an abstraction of its geometric structure such as point, line, or polygon. The location of an object indicates the position of the object with regard to the selected reference frame.

Topological features characterize relationships between spatial objects that refer to statements concerning adjacency, connectivity, inclusion, and similar relationships of objects. These relationships are independent from the used reference system and invariant under topological transformations like rotation and scaling. Also, spatial objects can be subject to temporal changes and thus be dynamic.

A spatial relationship is a relationship between two or more spatial objects. They include spatial predicates which compare two spatial objects with respect to some spatial relationship. They conform to traditional binary relationships and return a boolean value.

Topological relationships are the most formally investigated spatial relationships. They use topological properties for their description and include concepts like *continuity*, *adjacency*, *overlapping*, *interior*, *boundary*, *connectivity*, and *inclusion*. An essential property is that they are preserved under topological transformations such as rotation and scaling.

Metric relationships use measurements such as *distances* and *directions*. Spatial order relationships are based on the definition of order. As a rule, each order relation has an inverse relationship. For instance, *behind* is a spatial order relationship based on the order of reference with the inverse relationship in_front_of. Since the directional relationships are influenced by the relative size, distance, and the shapes of the two objects, so they are considered as fuzzy concepts.

Spatial objects are manipulated by spatial operations. A spatial operation is defined as a function with spatial arguments, i.e., it takes spatial objects as operands and returns either spatial objects or scalar values as results.

Some spatial operations compute a metric property of a spatial object and return a number. The *area* operation returns the area and the *perimeter* operation returns the perimeter of a polygon object. The operation *length* calculates the total length of a line object.

On the other hand, some spatial operations returning atomic spatial objects or sets of objects, such as *intersection*. The intersection operation does not have

closure properties. For example, the intersection of two lines may be either a point or a line.

Another important class of operations are *spatial selection* and *spatial join* operations, which combine sets of spatial objects and compare spatial objects by spatial predicates. Given a set S of spatial objects, a spatial selection filters out all those objects of S that fulfill a selection condition given either by a spatial predicate or by a comparison expression. Given two sets S and T of spatial objects, a spatial join constructs new spatial objects that aggregates objects from S and T. The decision on whether a pair of spatial objects of S and T belongs to the result depends on a boolean expression in the join condition, which is either a spatial predicate or a comparison expression.

4.2 Triangulation

Traditionally, applications with spatial data are based upon coordinates. However, with coordinate geometry, the complexity of standard operations and the difficulty of guaranteeing that no geometric inconsistencies are overlooked. In particular, objects separated into non-coherent parts causes problems which are difficult to treat.

In [24] and [98], a general spatial data model based upon *simplicial complexes* is presented. A simplex is either a point, a finite straight line segment, or a

triangle. To be more precise, a 0-*simplex* is a set consisting of a single point in the Euclidean plane; an 1-*simples* is a set consisting of all the points on a straight line between two distinct points in the Euclidean plane, including the end points; and a 2-*simplex* is a set consisting of all the points on the boundary and in the interior of a triangle whose vertices are three non-collinear points.

A simplicial complex is a collection of non-overlapping simplexes, such that if a simplex belongs to the complex then so do all its components. A simplicial complex is uniquely determined by its maximal component simplexes.

Many algorithms of triangulating polygon have been proposed in the past [30, 29, 21, 73]. An arbitrary n-vertex polygon can be triangulated into $n - 2$ triangles in time $O(n \log^* n)$ [73]. This algorithm will be described in Section 4.2.1.

In this research, we extend the concept of simplex to *counterclockwise directed triangles*. The virtue of counterclockwise directed triangles lies in the simplicity to test whether a point is *inside* a polygon (a set of triangles) while the point-location problem represents a major computational geometry task and is the basis to determine spatial relationships between spatial objects.

Definition 1: A triangle is *counterclockwise directed* if its three vertexes, $\text{point}_1(x_1, y_1)$, $\text{point}_2(x_2, y_2)$, and $\text{point}_3(x_3, y_3)$ are counterclockwise orientated, i.e.,

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} > 0$$

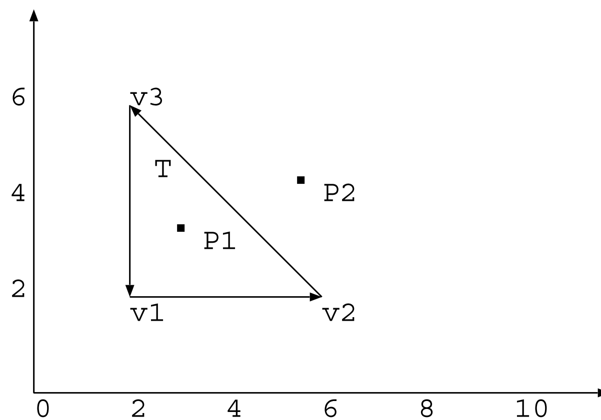

Figure 4.1: An example of counterclockwise directed triangle

Figure 4.1 is an example of a counterclockwise directed triangle. The vertexes of the triangle T — v_1, v_2 and v_3 are *counterclockwise* orientated. The edges $v_1 \to v_2$, $v_2 \to v_3$ and $v_3 \to v_1$ are directed lines. To test whether a point is *inside* a triangle is the same as to test whether a point is on the *left* side of all three edges of the triangle. For example, point P_1 is on the *left* side of all three edges of T, so P_1 is *inside* T. On the other hand, point P_2 is on the *left* side of edges $v_1 \to v_2$ and $v_3 \to v_1$ but not on the *left* side of edge $v_2 \to v_3$, so P_2 is not *inside* T.

4.2.1 Algorithm of Polygon Triangulation

Polygon triangulation has been studied extensively in Computational Geometry. Garey et al. [30] were the first to publish an $O(n \log n)$ algorithm. Then, another algorithm with the same complexity was published by Chazelle [12]. The $O(n \log n)$ bound was then improved to bounds of the form $O(n \log C_P)$, where C_P is a "shape" parameter no bigger than n that depends on the polygon P to be triangulated. Later, Tarjan and Van Wyk [88] made a major breakthrough with an $O(n \log \log n)$ algorithm, followed by Clarkon et al. [22], with a randomized algorithm with $O(n \log^* n)$ expected running time. In 1991, Seidel [73] presented a simpler randomized algorithm also with $O(n \log^* n)$ expected running time. In this section, we will focus on the algorithm developed by Seidel [73], one of the fastest triangulation algorithms and also the easiest to implement [62].

Consider a set S of n non-horizontal, non-crossing line segments. Starting at each endpoint of each segment in S, draw two horizontal rays, one towards the left and one towards the right, each extending until it hits a segment of S. For a segment endpoint p, the union of these two possibly truncated rays emanating from p is called the horizontal extension through p. The trapezoidations (Figure 4.2) of S, or $\Gamma(S)$ for short, is the segments of S together with horizontal extensions through the endpoints.

Let $S = \{s_0, s_1, \ldots, s_{n-1}\}$ be the set of edges around a simple polygon P,

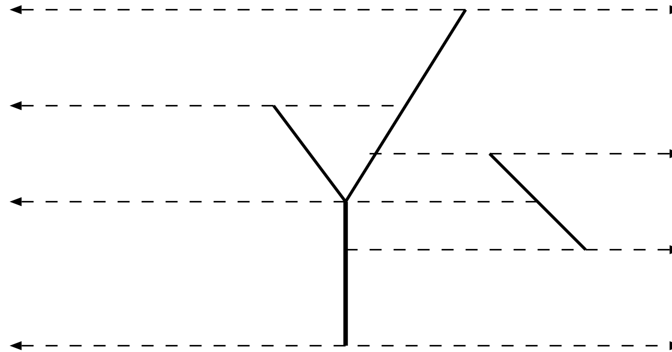

Figure 4.2: A trapezoidation of 4 line segments

assume that no two vertices of P have the same y-coordinate. This condition can always be achieved by rotating the coordinate system by an sufficiently small amount, thus the assumption will not cause loss of generality. A triangulation of P from a trapezoidation $\Gamma(S)$ is computed in two steps: (1) remove from consideration all trapezoids of $\Gamma(S)$ that do not lie in the interior of P; (2) for each of the remaining trapezoids check whether it has two vertices of P on its boundary that do not lie on the same side. If such a pair of vertices exists, draw a diagonal between them. The diagonals introduced in the second step partition P into a number of subpolygons, each of which has a very special form: its boundary consists of two y-monotone chains, one of which is a single edge. A polygon of such a form can easily be triangulated in linear time by repeatedly "cutting off" convex corners of the y-monotone chain.

Figure 4.3 is an example of triangulating a polygon. In (a), a trapezoidaized polygon is showed. Next in (b), all the trapezoids not inside the polygon are

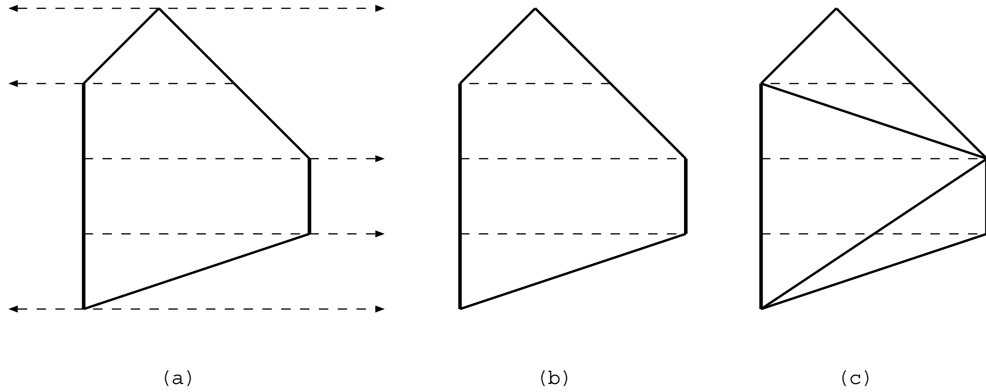

Figure 4.3: (a) A polygon; (b) Trapezoids inside the polygon; (c) Introducing diagonals

removed. Then in (c), diagonals are added, the original polygon has been divided into 3 triangles.

The complete algorithm is as follows.

Let $\log^{(i)} n$ denote the ith iterated logarithm, i.e., $\log^{(0)} n = n$ and for $i > 0$ we have $\log^{(i)} n = \log(\log^{(i-1)} n)$. For $n > 0$ let $log^{*}n$ denote the largest integer l so that $\log^{(l)} n \geq 1$, and for $n > 0$ and $0 \leq h \leq \log^{*} n$, let $N(h)$ be shorthand for $\lceil \frac{n}{\log^{(h)} n} \rceil$. Given trapezoidation $\Gamma(S)$, where S is a set of edges of a simple polygon, $\Psi(S)$ is a point location query structure for $\Gamma(S)$, i.e., $\Psi(S)$ is a directed acyclic graph with one source and with exactly one sink for each trapezoid of $\Gamma(S)$.

The input to the algorithm below is a simple polygonal chain C of n segments in consecutive order along C.

1. Generate s_1, s_2, $\ldots$, s_n, a random ordering of the segments of C

2. Generate Γ_1, the trapezoidation for the set $\{s_i\}$ along with the corresponding search structure Ψ_1

3. For $h = 1$ to $\log^* n$ do

   (a) For $N(h-1) < i \leq N(h)$ do

      i. Obtain trapezoidation Γ_i and search structure Ψ_i from Γ_{i-1} and Ψ_{i-1} by inserting segment s_i

   (b) Trace C through $\Gamma_{N(h)}$ to determine for each endpoint of all non-inserted segments the containing trapezoid of $\Gamma_{n(h)}$

4. For $N(\log^* n) < i \leq n$ do

   (a) Obtain trapezoidation Γ_i and search structure ψ_i from Γ_{i-1} and Ψ_{i-1} by inserting segment s_i

4.3 Spatial Relationships Between Triangles

The spatial relationships between two triangles are defined as follows:

- **equal(triangle$_1$, triangle$_2$)** — iff the set of the vertexes of triangle$_1$ are equal to the set of the vertexes of triangle$_2$

- **overlap(triangle$_1$, triangle$_2$)** — iff at least one edge of triangle$_1$ crosses with an edge of triangle$_2$

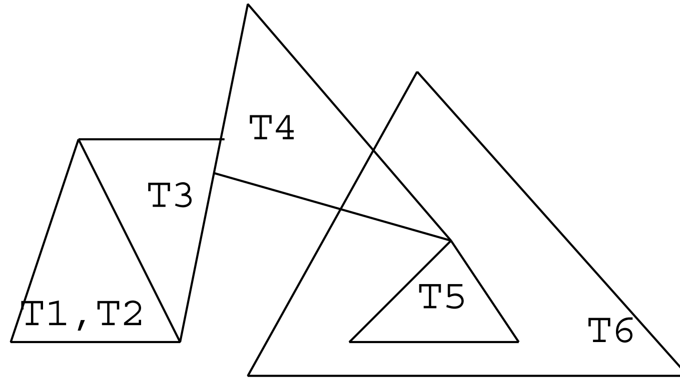

Figure 4.4: Example of Relationships Between Triangles

- **contain(triangle$_1$, triangle$_2$)** — iff the three vertexes of triangle$_2$ are all inside triangle$_1$

- **disjoint(triangle$_1$, triangle$_2$)** — iff non of the vertexes of triangle$_1$ is inside triangle$_2$ and vice versa; and non of the edges of triangle$_1$ crosses with any edge of triangle$_2$

- **adjacent(triangle$_1$, triangle$_2$)** — iff one edge of triangle$_1$ overlaps with an edge of triangle$_2$

- **commonborder(triangle1, triangle2)** — iff two vertexes of triangle$_1$ are equal to two vertexes of triangle$_2$

- **meet(triangle$_1$, triangle$_2$)** — iff one vertex of triangle$_1$ is on an edge of triangle$_2$

Figure 4.4 illustrates the definition of these relationships between two triangles. For example, (i) $T1$ and $T2$ which have the same set of vertexes are *equal* to each other. (ii) $T1$ and $T3$ share a *commonborder*, so do $T2$ and $T3$; (iii) $T1$, $T2$, and $T3$ are *disjoint* with $T5$ and $T6$, also $T1$ and $T2$ are *disjoint* with $T4$; (iv) $T3$ and $T4$ are *adjacent* because an edge of $T3$ overlaps with an edge of $T4$ and two vertexes of $T3$ are not inside $T4$; (v) $T4$ *meets* $T5$ because one vertex of $T4$ is on an edge of $T5$ and the other two vertexes of $T4$ are not inside $T5$; (vi) $T6$ *overlaps* $T4$ because one vertex of $T4$ is inside $T6$; and (vii) $T6$ *contains* $T5$ because all three vertexes of $T5$ are inside $T6$.

Furthermore, the relationships between these spatial operators are:

$$\begin{array}{ccc} \mathsf{equal} & \Longrightarrow & \mathsf{contain} \\ \mathsf{equal} & \Longrightarrow & \mathsf{overlap} \\ \mathsf{contain} & \Longrightarrow & \mathsf{overlap} \\ \mathsf{adjacent} & \Longrightarrow & \mathsf{overlap} \\ \mathsf{commonborder} & \Longrightarrow & \mathsf{overlap} \\ \mathsf{meet} & \Longrightarrow & \mathsf{overlap} \\ \mathsf{commonborder} & \Longrightarrow & \mathsf{adjacent} \end{array}$$

For example, if equal(triangle1, triangles2) evaluates to TRUE, then it implies that contain(triangle1, triangle2) also evaluates to TRUE.

4.4 Spatial Operations on Points, Lines and Triangles

The operations associated with points, lines and triangles are as follows:

- **distance(point**$_1(x_1, y_1)$**, point**$_2(x_2, y_2)$**)** $= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

- **inside(point**(x_0, y_0)**, line**$((x_1, y_1), (x_2, y_2))$**)** — tests whether the point is on the line, i.e.,

  $$\begin{vmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} = 0$$

  and $(min(x_1, x_2) \leq x_0 \leq max(x_1, x_2)$ or $min(y_1, y_2) \leq y_0 \leq max(y_1, y_2))$

- **distance(point**(x_0, y_0)**, line**$((x_1, y_1), (x_2, y_2))$**)**

  $$= \begin{cases} \dfrac{\begin{vmatrix} x_0 - x_1 & y_0 - y_1 \\ x_2 - x_1 & y_2 - y_1 \end{vmatrix}}{\sqrt{(y_2-y_1)^2+(x_2-x_1)^2}} \\ \quad if \begin{vmatrix} y_2 - y_1 & x_2 - x_1 \\ x_1 - x_0 & y_1 - y_0 \end{vmatrix} \times \begin{vmatrix} y_2 - y_1 & x_2 - x_1 \\ x_2 - x_0 & y_2 - y_0 \end{vmatrix} \leq 0 \\ min(\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}, \sqrt{(x_0 - x_2)^2 + (y_0 - y_2)^2}) \\ \quad otherwise \end{cases}$$

- **center_of_mass(line**$((x_1, y_1), (x_2, y_2))$**)** — the x and y coordinates of the center of mass of a line is:

$$cmx = \frac{x_1 + x_2}{2}, cmy = \frac{y_1 + y_2}{2}$$

- **length(line** $((x_1, y_1), (x_2, y_2))$**)** $= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

- **crosspoint(line$_1$,line$_2$)** — the cross point of two lines is point$_c$ (x, y) with the coordinates as:

$$x = \frac{(x_1 - x_2)\begin{vmatrix} x'_1 & y'_1 \\ x'_2 & y'_2 \end{vmatrix} - (x'_1 - x'_2)\begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix}}{(y_1 - y_2)(x'_1 - x'_2) - (y'_1 - y'_2)(x_1 - x_2)}$$

$$y = \frac{(y_1 - y_2)\begin{vmatrix} x'_1 & y'_1 \\ x'_2 & y'_2 \end{vmatrix} - (y'_1 - y'_2)\begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix}}{(y_1 - y_2)(x'_1 - x'_2) - (y'_1 - y'_2)(x_1 - x_2)}$$

  and $(min(x_1, x_2) \leq x \leq max(x_1, x_2)$ or $min(y_1, y_2) \leq y \leq max(y_1, y_2))$.

  When there is no solution to the above equations, we say point$_c$ $= null$;

- **intersect(line$_1$, line$_2$)** — if the cross point of the two lines is not *null*, we say line$_1$ and line$_2$ cross with each other

- **center_of_mass(triangle**$((x_1, y_1), (x_2, y_2), (x_3, y_3))$**)** — the x and y coordinates of the center of mass of a triangle is:

$$cmx = \frac{x_1 + x_2 + x_3}{3}, cmy = \frac{y_1 + y_2 + y_3}{3}$$

- **area(triangle**$((x_1, y_1), (x_2, y_2), (x_3, y_3))$**)** $= \frac{1}{2} \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$
- **perimeter(triangle**$((x_1, y_1), (x_2, y_2), (x_3, y_3))$**)** $= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ $+ \sqrt{(x_3 - x_2)^2 + (y_3 - y_2)^2} + \sqrt{(x_1 - x_3)^2 + (y_1 - y_3)^2}$
- **intersect(line, triangle)** — returns a segment of the line that is inside the triangle, which may be also be a point.

  To calculate the intersection of a line and a triangle, first test if any end-points of the line is inside the triangle. If both of the endpoints are inside the triangle, then return the line. Next, calculate the cross point of the line and the edges of the triangle. If there are two such points, then return the line segment with these two points as endpoints; if there is only one cross point and one endpoint of the original line is inside the triangle, then return the line segment with the cross point and the original endpoint as endpoints; if there is only one cross point and there is no endpoint of the original line inside the triangle, then return the cross point.
- **intersect(triangle$_1$, triangle$_2$)** — if two triangles intersect with each other, it may return a set of triangles, a line segment or only a point.

Algorithm 2 is a simplified version of the algorithm of intersection of convex polygons presented in [67]. In this algorithm, L is a list containing the coordinates of points, which is empty at the beginning of the algorithm. Each cross point of the edges of the two triangles are appended to L, and the vertex of the triangles is also appended to L if it is inside the other triangle. Then L is triangulated using the algorithm in Section 4.2.1. If L only contains one element, that means the two triangles meet at one point; if L contains two element, that means the two triangles are adjacent with each other.

Algorithm 2 intersect two triangles into a region

Require: initially T_1, T_2, $L = \emptyset$.

1: **for** each edge E of T_1 **do**
2:     **for** each edge E' of T_2 **do**
3:         $point_c = \mathsf{crosspoint}(E, E')$
4:         **if** $point_c \neq null$ **then**
5:             append $point_c$ to L
6:         **end if**
7:     **end for**
8: **end for**
9: **for** each vertex V of T_1 **do**
10:     **if** V inside T_2 **then**
11:         append V to L
12:     **end if**
13: **end for**
14: **for** each vertex V' of T_2 **do**
15:     **if** V' inside T_1 **then**
16:         append V' to L
17:     **end if**
18: **end for**
19: triangulate L into set of triangles S
20: return S

4.5 Spatial Relationship Between Polygons

The spatial operators defined for triangles in Section 4.3 can be used towards polygons with little change.

Let S and S' denote sets of triangles that two polygons R_1 and R_2 are decomposed into, and t and t' denote an element in S and S', respectively. The operators proposed in Section 4.3 can be used on two polygons in the following ways:

1. R_1 is `equal` to R_2 if $S = S'$.
2. R_1 `overlaps` R_2 if $\exists t \in S$, $t' \in S'$, $\exists$edge e of t, edge e' of t', such that e and e' intersect
3. R_1 `contains` R_2 if $\forall t' \in S'$, $\forall$vertexes v of t', $\exists t \in S$, such that v is *inside* t.
4. R_1 is `disjoint` with R_2 if $\forall t \in S$, $\neg \exists t' \in S'$, such that t *overlaps* t'.
5. R_1 is `adjacent` with R_2 if $\exists t \in S$, $\exists t' \in S'$ where t is *adjacent* with t' and the two ends of the adjacent edge are *neighboring vertexes* of both R_1 and R_2.
6. R_1 and R_2 have a `commonborder` if $\exists t \in S$, $\exists t' \in S'$ where t and t' have a *common border* and the two ends of the common border are *neighboring*

vertexes of both R_1 and R_2.

7. R_1 `meets` R_2 if $\exists t \in S$, $\exists t' \in S'$, such that t *meets* t' and $\forall t'' \in S'$, t'' does not *contain* t.

Since a polygon with n vertexes can be decomposed into $n-2$ triangles while the relationships between two triangles can be determined in constant time, so the comparison of two polygons can be done in $O(n^2)$ time.

4.6 Summary

In this chapter, we reviewed the characteristics of spatial data and extends the *simplexes* concept to *counterclockwise directed triangles*, which is used together with points and lines to model spatial data.

This chapter also introduced the polygon triangulation algorithm developed by Seidel [73], which can be done in $O(n \log^* n)$ time.

The spatial relationships between triangles and spatial operations on points, lines and triangles were then presented. Finally, we discussed how to define spatial relationships between two polygons based on the relationships between triangles.

CHAPTER 5

A Concrete Model of Spatio-Temporal Data

This chapter presents the design and implementation of SQL^{ST}, an extensible spatio-temporal data model and query language. We use counterclockwise directed triangles to model spatial objects and intervals to model time. In addition to providing the spatio-temporal primitives as built-ins, SQL^{ST} allows the user to introduce additional extensions to the data model and query language. The two main mechanisms used for this purpose are user-defined aggregates and table expressions. This approach minimizes the extensions required in SQL, supports the orthogonality of spatial and temporal constructs, and the customization of spatio-temporal extensions to meet the different needs of different applications.

5.1 SQL^{ST}

We now introduce SQL^{ST} through examples. For concreteness, we consider a spatio-temporal application from the NSF Arctic System Science (ARCSS) Research Program Ocean Atmosphere Ice Interactions (OAII) Project. The marine

environment is an interactive system comprising the water, ice, air, biota, dissolved chemicals and sediments. ARCSS/OAII seeks to enhance understanding of this system and its role in climate and global change. The extratropical cyclone data set [4] at ARCSS provides information about cyclones in the Northern Hemisphere. We will use this data set through out our study.

Consider a cyclone database at a weather center. Included in this database is the information about cyclone activities including the path and the center of pressure for cyclones during time intervals.

Example 35 *Define the* Cyclone *relation.*

```
CREATE TABLE Cyclone
          (ID INT, Pressure REAL, Trajectory LINE,
          Tstart DATE, Tend DATE)
```

In this relation, the Tstart and Tend columns indicate the start and end instants of a time interval, and have a granularity of one day. Trajectory is a line segment whose start and end points are the positions[1] of the cyclone at time Tstart and Tend, respectively. Pressure is the average of the pressure values measured at the center of the cyclone at Tstart and Tend, and its unit is *milibar*. ID is the cyclone identifier. ID and Tstart together serve as the key of the relation.

[1] $x = R_{earth}\frac{longitude}{180}\pi$, $y = R_{earth}\frac{latitude}{180}\pi$

A second type of information contained in the database is about regions of interest. For instance, all the islands in an archipelago might be described as follows.

Example 36 *Define the* Island *relation.*

```
CREATE TABLE Island
        (Name CHAR(30), Region TRIANGLE)
```

In this relation, the Region column stores the geometry of the island. Each island is decomposed into non-overlapping triangles. So our table contains a row with Name = "Maui" for each triangle into which the region of Maui has been decomposed. In general, polygonal regions are represented as sets of non-overlapping, counterclockwise directed triangles.

Our basic objects are *points*, *lines*, and *counterclockwise directed triangles*; thus, a polygon is represented by a set of counterclockwise directed triangles. By decomposing polygons into sets of counterclockwise directed triangles, the hard-to-express spatial relationships between two spatial objects can be evaluated easily [17]. The algorithms to decompose a polygon into directed triangles and to merge triangles into polygons can be found in [73, 62]. Finally, we use *time intervals* to model temporal data at the physical level.

A set of typical cyclone-related queries for NFS ARCSS/OAII dataset is given

next; we had also used similar queries to analyze the computer simulation data produced by a climatic weather model on a NASA-sponsored project [59].

Example 37 *Find all cyclones whose high pressure stage (pressure > 1000mb) lasted more than 3 days.*

```
SELECT ID
FROM Cyclone
WHERE Pressure > 1000
GROUP BY ID
HAVING DURATION(Tstart, Tend) > 3
```

This query will return the IDs of the cyclones whose high pressure stage last more than 3 days.

As it can be easily inferred from the syntactic structure of this query that uses the GROUP BY and HAVING constructs, the operator duration is in fact a user-defined aggregate. The definition and implementation of this aggregate in SQL^{ST} will be discussed in detail in Section 5.2. Similar comments apply to all the remaining queries in this section.

Example 38 *Find the cyclones whose trajectory have been enclosed by Maui.*

```
SELECT ID
FROM Cyclone, Island
```

```
WHERE Name = "Maui"
GROUP BY ID
HAVING CONTAIN(Region, Trajectory)
```

This query returns the ID of the cyclones have been completely inside the island "Maui".

Example 39 *Find how long each cyclone has traveled when it was over Maui.*

```
SELECT ID, SUM(length(intersect(Trajectory, Region)))
FROM Cyclone, Island
WHERE Name = "Maui"
GROUP BY ID
HAVING OVERLAP(Trajectory, Region)
```

This query returns the distance the cyclones traveled when their trajectories overlap the region of the island "Maui".

The functions intersect and length are built-in C++ functions described in Section 5.2.1. In this case, when a line (Trajectory) overlaps with a triangle (Region), their intersection will be a line segment. The length of the line segments is then calculated and summed up.

Other spatial relationships similar to contain and overlap include adjacent, disjoint, and equal, etc. All these relationships can be evaluated using user-defined

aggregates.

Example 40 *Identify all cyclones that have come within 50 miles of the coast of Lanai.*

```
SELECT ID
FROM Cyclone, Island
WHERE Name = "Lanai"
GROUP BY ID
HAVING EDGE_DISTANCE(Trajectory, Region) <= 50
```

This query finds the cyclones whose shortest distance from their trajectories to the coast of the island "Lanai" is less than or equal to 50 miles.

Example 41 *Find the cyclones that have traveled for more than 300 miles continuously.*

```
SELECT ID
FROM Cyclone
GROUP BY ID
HAVING MOVING_DISTANCE(Trajectory, Tstart, Tend) > 300
```

This query returns the IDs of the cyclones whose center have moved continuously over 300 miles.

In the above examples, we see SQL^{ST} is a natural minimalist extension to SQL.

5.2 Implementation

5.2.1 Built-in Functions

As the starting point of our SQLST system, we implemented the following built-in operators in C++. Their definition has been discussed in Chapter 4.

- **distance(point$_1(x_1, y_1)$, point$_2(x_2, y_2)$)**
- **inside(point, line)** — returns 1 if the point is *on* the line; 0 otherwise.
- **distance(point(x_0, y_0), line$((x_1, y_1), (x_2, y_2))$)**
- **center_of_mass(line$((x_1, y_1), (x_2, y_2))$)**
- **length(line$((x_1, y_1), (x_2, y_2))$)**
- **intersect(line$_1$, line$_2$)** — returns their common segment (which may be a point) if any; return $((0, 0), (0, 0))$ otherwise.
- **inside(point, triangle)** — returns 1 if the point is on the *left* side of all three edges of the triangle; 0 otherwise.
- **center_of_mass(triangle)**
- **area(triangle)**
- **intersect(line, triangle)** — returns a segment of the line that is inside the triangle if any; return $((0, 0), (0, 0))$ otherwise.

- **intersect(triangle$_1$, triangle$_2$)** — return their common area as a set of triangles (which may also be a point or a line segment) if any; return $((0,0),(0,0),(0,0))$ otherwise.

These functions are a partial list of the spatial relationships and operations between points, lines and triangles that SQLST support. For more information, please refer to [17].

Based on these built-in functions, users can define spatial relationships between two spatial objects such as contain, overlap and edge_distance, etc.

5.2.2 AXL

The AXL (Aggregate eXtension Language) system, introduced in [96], implements the SQL language on top of Berkeley DB record manager [79]. But in addition to supporting SQL, and allowing the users to introduce new functions by programming them in C/C++ (as O-R systems do), AXL supports the definition of powerful user-defined aggregates (UDAs) expressed in an SQL-like language. By programming new UDAs in SQL, end-users can easily extend the database system and support a variety of advanced database applications [96]. Applications supported through AXL's UDA extensions include, OLAP, data mining, temporal and spatial databases, and others that make extensive use of data aggregation. Furthermore, AXL is efficient, and only a limited overhead is paid for

developing such applications via UDAs defined in SQL, rather than directly in a procedural language such as C/C++.

To create an aggregate in AXL, a user needs to define three SQL routines, under the labels of INITIALIZE, ITERATE and TERMINATE, which, respectively, specify the computation to be performed for the first value in the stream, for each successive value, and when the end of the stream is detected. Results of the aggregation can be returned anytime during these routines by inserting tuples into an append-only RETURN stream.

For instance, to create an online average aggregate which returns results (current averages) for every 100 new values, we can define the following aggregate.

Example 42 *MOVING_AVERAGE – a user-defined aggregate in AXL.*

```
AGGREGATE MYAVG(Next INT) : REAL
{
    TABLE state(sum INT, cnt INT);
    INITIALIZE : {
        INSERT INTO state VALUES(Next, 1);
    }
    ITERATE : {
        UPDATE state SET sum = sum + Next, cnt = cnt + 1;
        INSERT INTO return SELECT sum/cnt FROM state WHERE cnt%100 = 0;
```

```
    }
    TERMINATE : {
        INSERT INTO return SELECT sum/cnt FROM state;
    }
}
```

The first line of this aggregate function declares a local table, STATE, to keep (in memory) the sum and count of the values processed so far. While, for this particular example, STATE contains only one tuple, it is in fact a table that can be queried and updated using SQL statements. These SQL statements are grouped into the three blocks labelled respectively INITIALIZE, ITERATE and TERMINATE. Thus, INITIALIZE inserts the value taken from the input stream and sets the count to 1. The ITERATE statements update the table by adding the new input value to the sum and 1 to the count. The TERMINATE statements return the final result of computation by appending it to RETURN; for conformity with SQL, RETURN is viewed as a table, and thus an INSERT INTO construct is used. We also add intermediate results from the computation to RETURN tables as part of the ITERATE statements.

The fact that UDAs in AXL are written in SQL achieves compatibility of data types and programming paradigms and inherits the well-known benefits of database query languages, such as scalability, data independence and paralleliz-

ability. The ability of introducing and manipulating new tables is one of the first cornerstones of AXL's power. The second is the ability of one aggregate calling another, or calling itself recursively.

5.2.2.1 Implementation of AXL

The AXL compiler translates AXL programs into C++ code. AXL adopts an open interface for its physical data model, so that the system can link with a variety of physical database implementations. The Berkeley DB library [79] is now used as AXL's main storage manager, but we have now added support for in-memory tables, and there is current work to support R-trees [38].

The runtime model of AXL is based on data pipelining. In particular, all UDAs, including recursive UDAs that call themselves, are pipelined, which means tuples inserted into the RETURN relation during the INITIALIZE/ITERATE steps are returned to their caller immediately. In order to do this, all local variables (temporary tables) declared inside the body of a UDA are assembled into a STATE structure which is passed into the UDA for each INITIALIZE/ITERATE/TERMINATE call.

AXL UDAs can either be used as stand-alone programs or, imported into O-R systems such as DB2 (with limitations due to the fact that we use UDFs that return a single value for each call).

5.2.3 Temporal Aggregates

We implemented our SQLST system by first adding the built-ins described in Section 5.2.1 into AXL, and then using its UDA mechanism to implement the spatio-temporal aggregates needed in spatio-temporal queries such as those of Section 5.1. For instance, the temporal aggregate duration is implemented as follows:

Example 43 *DURATION*

```
AGGREGATE DURATION(Tstart DATE, Tend DATE) : INT
{
    TABLE state(i INT);
    INITIALIZE : {
        INSERT INTO state VALUES(Tend - Tstart + 1);
    }
    ITERATE : {
        UPDATE state SET i = i + (Tend - Tstart + 1);
    }
    TERMINATE : {
        INSERT INTO return SELECT i FROM state;
    }
}
```

This aggregate calculates the total length of the time intervals. For example, if we have a set of tuples as

```
cyclone(930001, _, _, '1993-01-01', '1993-01-05')
cyclone(930001, _, _, '1993-01-11', '1993-01-15')
cyclone(930001, _, _, '1993-01-21', '1993-01-25')
```

The result of duration will be 15 days.

5.2.4 Spatial Aggregates

Similar to the temporal aggregates, spatial relationships can also be expressed using user-defined aggregates. We also take advantage of inheritance and overloading characteristics of O-R systems [86]. Both point and line are subtypes of triangle. A point can be considered as a triangle whose three vertexes are the same and a line as a triangle whose first two vertexes are the two endpoints of the line and the last vertex is the center of mass of the line.

Example 44 *EDGE_DISTANCE*

```
AGGREGATE EDGE_DISTANCE(Object1 TRIANGLE, Object2 TRIANGLE) : REAL
{
```

```
    TABLE state(d REAL);
    INITIALIZE : ITERATE : {
        INSERT INTO state VALUES(distance(Object1.Vertex, Object2.Edge));
    }
    TERMINATE : {
        INSERT INTO return SELECT MIN(d) FROM state;
    }
}
```

As mentioned in Section 5.2.1, the built-in function distance(point,line) calculates the distance between a point and a line segment. The *edge_distance* between two spatial objects, which are two sets of triangles, is defined as the smallest distance between the vertexes of the triangles of one set to the edges of the triangles of the other set.

Example 45 *CONTAIN*

```
AGGREGATE CONTAIN(Object1 TRIANGLE, Object2 TRIANGLE) : INT
{
    TABLE state(b INT) AS VALUES(1);
    TABLE triangles(Object TRIANGLE);
    TABLE points(Vertex POINT);
    INITIALIZE : ITERATE : {
```

```
        INSERT INTO triangle VALUES(Object1);
        INSERT INTO points VALUES(Object2.Vertex);
    }
    TERMINATE : {
        UPDATE state SET b = 0 WHERE NOT EXIST
            (SELECT Vertex FROM points, triangles
            WHERE inside(Vertex, Object) = 1);
        INSERT INTO return SELECT b FROM state WHERE b = 1;
    }
}
```

The function inside is a built-in function that will evaluate to 1 if a point is *inside* a triangle; 0 otherwise.

The spatial aggregate contain($Object_1$, $Object_2$) then uses inside to test if $Object_1$ contains $Object_2$. The aggregate first iterate through all the records and insert the triangles belong to $Object_1$ into one table and the vertexes belong to $Object_2$ into another table. If there exists one vertex of $Object_2$ that is not inside any triangle of $Object_1$, then the aggregate contain will not return anything; otherwise, it will return 1, i.e., contain($Object_1$, $Object_2$) $\equiv \forall$ triangle t' $\in Object_2$, $\forall$ vertex v of t', $\exists$ triangle t $\in Object_1$, v inside t.

Another example is the aggregate overlap, which can be defined as follows.

Example 46 *OVERLAP*

```
AGGREGATE OVERLAP(Object1 TRIANGLE, Object2 TRIANGLE) : INT
{
  TABLE state(b INT) AS VALUES(0);
  TABLE edges1(line1 LINE);
  TABLE edges2(line2 LINE);
  INITIALIZE : ITERATE : {
    INSERT INTO edges1 VALUES(Object1.Edge);
    INSERT INTO edges2 VALUES(Object2.Edge);
  }
  TERMINATE : {
    UPDATE state SET b = 1 WHERE EXIST
       (SELECT line1 FROM edges1, edges2 WHERE intersect(line1,line2) = 1);
    INSERT INTO return SELECT b FROM state WHERE b = 1;
  }
}
```

The function intersect is a built-in function that will evaluate to 1 if two lines cross with each other; 0 otherwise.

Similar to contain, the spatial aggregate overlap then uses intersect to test if any edge of a triangle belongs to Object_1 crosses with an edge of a triangle belongs to Object_2. Again, the aggregate first iterate through all the records and insert

the edges of the triangles into two different tables. If there exists a pair of edges of triangles belong to Object_1 and Object_2 that intersect with each other, then $\text{overlap}(\text{Object}_1, \text{Object}_2)$ will return 1; otherwise, it will not return anything, i.e., $\text{overlap}(\text{Object}_1, \text{Object}_2) \equiv \text{intersect}(\text{Object}_1.\text{Edge}, \text{Object}_2.\text{Edge}) \neq \emptyset$.

In a similar fashion, end-users can easily define additional operators such as adjacent, disjoint, thus producing a spatio-temporal database system that has powerful primitives and is customized for their application.

5.2.5 Spatio-Temporal Aggregates

Some operators may require dealing with both spatial and temporal information at the same time, such as moving_distance.

Example 47 *MOVING_DISTANCE*

```
AGGREGATE MOVING_DISTANCE(Object, Tstart DATE, Tend DATE) : REAL
{
  TABLE state(d REAL, x REAL, y REAL, time DATE);
  INITIALIZE : {
    INSERT INTO state VALUES(0, cmx(Object), cmy(Object), Tend);
  }
  ITERATE : {
    UPDATE state SET d = d + sqrt(cmx(Object) - x)^2 + (cmy(Object) - y)^2),
```

```
        x = cmx(Object), y = cmy(Object), time = Tend
        WHERE Tstart = time + 1);
      INSERT INTO state VALUES(0, cmx(Object), cmy(Object), Tend)
        WHERE Tstart ≠ time + 1);
    }
    TERMINATE : {
      INSERT INTO return SELECT d FROM state;
    }
}
```

This aggregate calculates the distance an object travels continuously. We first calculate the distance between two positions of the center of mass of an object at two consecutive time intervals, and then sum up the distance calculated. When the start of a new time interval is not continuous to the end of any existing time interval, we save the value and start a new round of calculation. The built-in functions cmx and cmy calculates the x and y coordinates of the center of mass of an object, respectively.

5.3 Performance

A key question to evaluate the effectiveness of the SQLST approach is how much overhead must be paid for having programmed our extensions in SQL rather

than in C or C++. Therefore we performed the following experiment on the cyclone data set obtained from [4]. The data set contains a 28-year record (1 May 1966 through 31 December 1993) of daily cyclone statistics for the Northern Hemisphere. The data set includes the position and central pressure of each cyclone, together with the ID of the cyclone and a date field. The data set has about 200,000 records and is over 14MB. Also, we used an island relation that contains 1000 tuples. Then, we compared the performance of four queries written in C++ (accessing the data through the Berkeley-DB API), against those written in AXL. An index has been created on the Tstart columns of the Cyclone table and on the Name column of the Island table. We compare the results of the queries in four different cases: (1) AXL using indexes; (2) AXL not using indexes; (3) C++ using indexes; and (4) C++ not using indexes.

We tested the performance of the following four queries on a Pentium III with a single 500HZ processor and 160MB memory, running LINUX.

Example 48 *Find the duration of the cyclones occurred in June, 1966.*

```
SELECT ID, DURATION(Tstart, Tend)
FROM Cyclone
WHERE "1966-06-01" <= Tstart AND "1966-07-01" > Tstart
GROUP BY ID
```

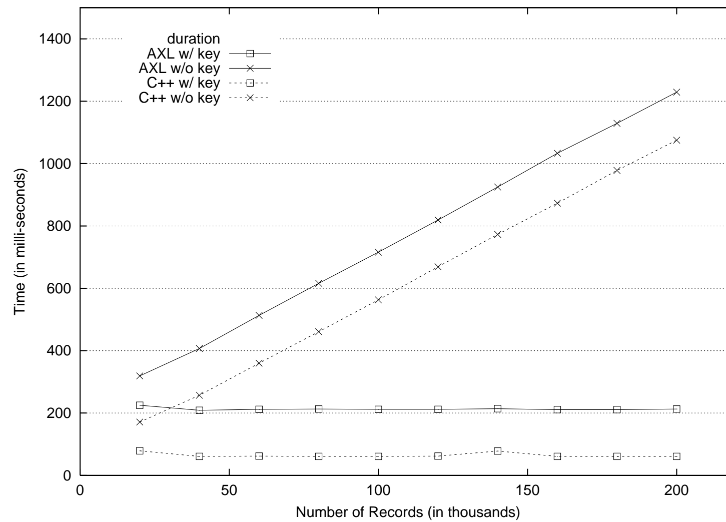

Figure 5.1: Performance result of Query 48

Example 49 *Find the distance traveled by the cyclones occurred in June, 1966.*

SELECT ID, MOVING_DISTANCE(Trajectory, Tstart, Tend)

FROM Cyclone

WHERE “1966-06-01” <= Tstart AND “1966-07-01” > Tstart

GROUP BY ID

Figures 5.1 and 5.2 are the results of the aggregates duration and moving_distance. Since the number of records that meet the selection criterion is fixed, so when a key is used, we only need to retrieve the qualified records directly, thus the result of performance of AXL and C++ are constant. When no key is used, we have to search the entire database, so the result of the performance of both AXL

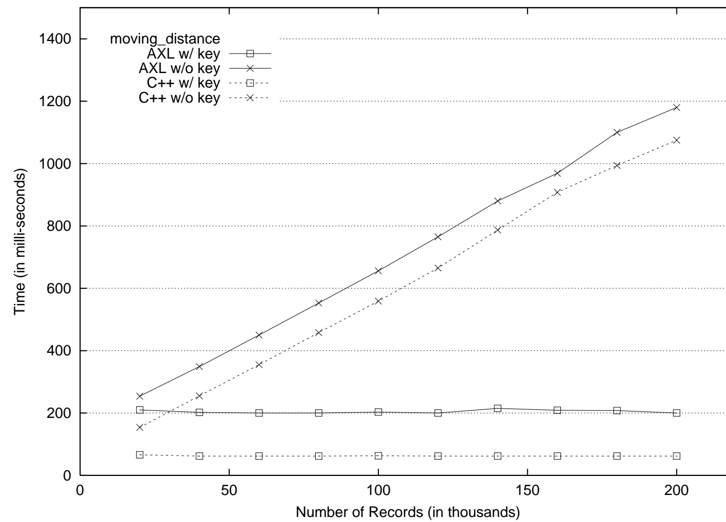

Figure 5.2: Performance result of Query 49

and C++ increase linearly. In both cases, the program written directly in C++ against the Berkeley DB API outperforms AXL by a constant.

Example 50 *Find the cyclones which occurred in June, 1966 and landed on the island Oahu.*

```
SELECT ID
FROM Cyclone, Island
WHERE "1966-06-01" <= Tstart AND "1966-07-01" > Tstart AND Name = "Oahu"
GROUP BY ID
HAVING CONTAIN(Region, Trajectory)
```

Example 51 *Find the distance between cyclones which occurred in June, 1966 and the coast of island Oahu.*

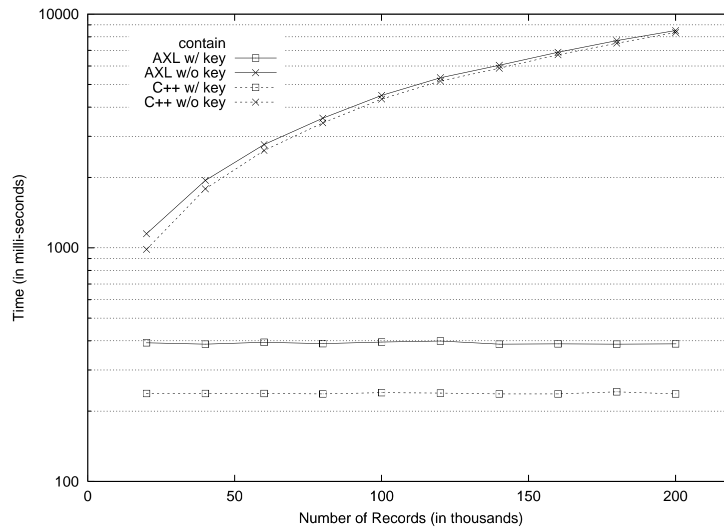

Figure 5.3: Performance result of Query 50

SELECT ID, EDGE_DISTANCE(Trajectory, Region)

FROM Cyclone, Island

WHERE "1966-06-01" <= Tstart AND "1966-07-01" > Tstart AND Name = "Oahu"

GROUP BY ID

Figure 5.3 and 5.4 are the results of the aggregates contain and edge_distance. Still, when a key is used, the results of performance of AXL and C++ are constant. When no key is used, the results of the performance of both AXL and C++ increase. Again, In both cases, C++ API of Berkeley-DB is slightly faster than AXL. We used *log* scale for the time axis in these two graphs because the difference between using a key or not using one is dramatic. Unlike the aggregates duration and moving_distance which handle the incoming data as soon as

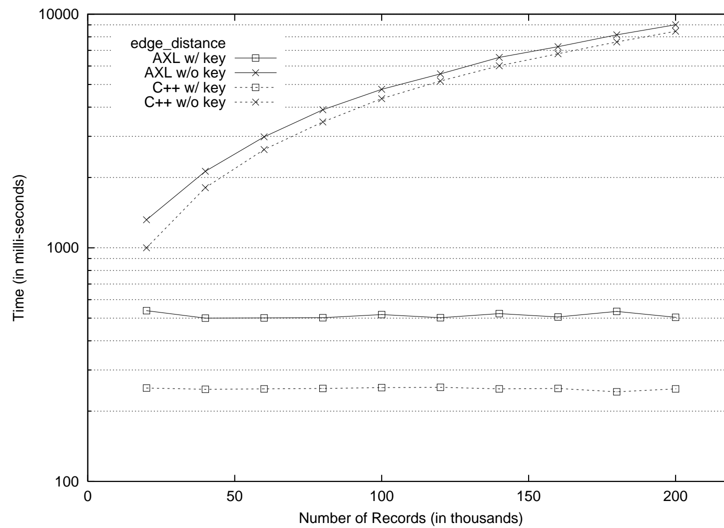

Figure 5.4: Performance result of Query 51

it comes in without storing it, the aggregates contain and edge_distance need to insert the incoming data into temporary tables first, and then upon termination, query these tables to get the desired results. Thus the performance curve of these two aggregates is different from that of duration and moving_distance.

5.4 More Abstract Representations

Spatio-temporal data modeling involves the abstraction of reality as a number of objects or features. The selection of abstract models depends on the application and users' preference.

To model time at the conceptual level, we may use a point-based time model [90], where information is repeated for each time granule where it is valid. The

polygon-oriented representation for spatial data is also selected because two dimensional shapes offer a more natural representation for many application domains. For instance, a region in a GIS system can be drawn, enlarged, moved, or split. Therefore, the abstract model of SQL^{ST} views reality as a sequence of snapshots of objects that are moving and/or changing in shape.

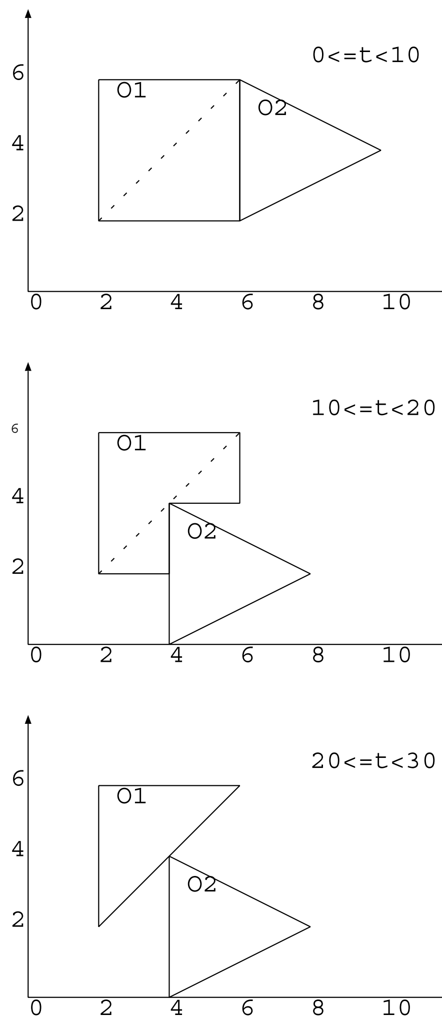

Figure 5.5: Graphs representing spatio-temporal data

Figure 5.5 is an example of spatial objects changing with time. At time $t = 0$, there are two spatial objects in the graph, a square $O1$ and a triangle $O2$. At time $t = 10$, $O1$ changes its shape and $O2$ is moved to a new position. At time $t = 20$, $O1$ has some more changes in shape while $O2$ stays unchanged.

Table 5.1 shows how the changes are recorded in the database at the abstract level. From time $t = 0$ on, the square $O1$ is represented by two triangles and the triangle $O2$ is represented by one triangle. At time $t = 10$, changes of the shape of $O1$ and position of $O2$ occur and their representation also changed accordingly. Now $O1$ is represented by three triangles while $O2$ is still represented by one triangle with new coordinates of the vertexes. This representation is valid from time $t = 10$ till further change occurs. At time $t = 20$, the shape of $O1$ is changed further while $O2$ remains unchanged. Now $O1$ is represented by one triangle and $O2$ stays unchanged.

In Table 5.1, we also notice that the valid time of each fact is recorded as a time instant VTime.

As we have discussed in the previous chapter, the internal representation of the spatio-temporal objects shown in Figure 5.5 is in Table 5.2, i.e., triangles are used for spatial data modeling and intervals are used for temporal data modeling.

A polygon having n vertexes in the abstract data model is decomposed into $n - 2$ triangles in the concrete data model. The time instants associated with the

ID	S	VTime
$O1$	[(2,2),(6,2),(6,6),(2,6)]	0
...	...	...
$O1$	[(2,2),(6,2),(6,6),(2,6)]	9
$O2$	[(6,2),(10,4),(6,6)]	0
...	...	...
$O2$	[(6,2),(10,4),(6,6)]	9
$O1$	[(2,2),(4,2),(4,4),(6,4),(6,6),(2,6)]	10
...	...	...
$O1$	[(2,2),(4,2),(4,4),(6,4),(6,6),(2,6)]	19
$O2$	[(4,0),(8,2),(4,4)]	10
...	...	...
$O2$	[(4,0),(8,2),(4,4)]	19
$O1$	[(2,2),(6,6),(2,6)]	20
...	...	...
$O1$	[(2,2),(6,6),(2,6)]	29
$O2$	[(4,0),(8,2),(4,4)]	20
...	...	...
$O2$	[(4,0),(8,2),(4,4)]	29

Table 5.1: Abstract model of the spatio-temporal data shown in Figure 5.5

ID	S	VTime
$O1$	((2,2), (6,2), (6,6))	[0, 10)
$O1$	((6,6), (2,6), (2,2))	[0, 10)
$O2$	((6,6), (6,2), (10,4))	[0, 10)
$O1$	((6,6), (2,6), (2,2))	[10, 20)
$O1$	((2,2), (4,2), (4,4))	[10, 20)
$O1$	((4,4), (6,4), (6,6))	[10, 20)
$O2$	((4,4), (4,0), (8,2))	[10, 20)
$O1$	((2,6), (2,2), (6,6))	[20, 30)
$O2$	((4,4), (4,0), (8,2))	[20, 30)

Table 5.2: Concrete model of the spatio-temporal data shown in Figure 5.5

same non-temporal value are coalesced into time intervals.

The extensibility mechanisms of SQLST can also be used to elevate the representation to richer and more abstract levels. Next, we show how SQLST can be further extended to support these more abstract representations. In fact, we will now express the queries of Section 5.1 using *points* and *polygons* as spatial data types and *time instants* as temporal data type.

5.4.1 Schema Definition

The schema of the databases in Section 5.1 have been changed as follows.

Example 52 *Define the* Cyclone *relation. (Example 35)*

```
CREATE TABLE Cyclone
        (ID INT, Pressure REAL, Position POINT, Time DATE)
```

In the Cyclone relation, the spatial data column is Position which has a data type point and specifies the x and y coordinates of the center of a cyclone. The temporal data column is Time which has a granularity of one day and captures the time instants of cyclones' movements.

Example 53 *Define the* Island *relation. (Example 36)*

```
CREATE TABLE Island
        (Name CHAR(30), Extent POLYGON(30))
```

In the Island relation, the Extent column has a spatial data type as polygon and specifies the geometry of the island.

5.4.2 Spatio-Temporal Queries

Since the spatio-temporal operators are supported at the internal level, so table expressions in O-R systems [13] are used to transform the queries.

For instance, Example 37 in Section 5.1 is expressed as follows at the conceptual level.

Example 54 *Find cyclones whose high pressure stage lasted more than 3 days (Example 37).*

```
SELECT New.ID
FROM (SELECT ID, MAPPING(Position, Time) FROM Cyclone
          WHERE C.Pressure > 1000 GROUP BY ID)
          AS New (ID, Trajectory, Tstart, Tend)
GROUP BY New.ID
HAVING DURATION(New.Tstart, New.Tend) > 3
```

Mapping is also a user-defined aggregate which maps a pair of point and time instant into a pair of line and time interval.

Example 55 *MAPPING*

```
AGGREGATE MAPPING(position POINT, time DATE) : (LINE, DATE, DATE)
{
    TABLE state(l LINE, d1 DATE, d2 DATE);
    TABLE tmp(p POINT, t DATE);
    INITIALIZE : {
        INSERT INTO tmp VALUES(position, time);
    }
    ITERATE : {
        INSERT INTO state VALUES(SELECT p, position, t, time FROM tmp
              WHERE time = t + 1);
        UPDATE tmp SET p = position, t = time;
    }
    TERMINATE : {
        INSERT INTO return SELECT l, d1, d2 FROM state;
    }
}
```

Similarly, Example 38 can be expressed as follows.

Example 56 *Find the cyclones whose trajectory have been enclosed by Maui (Example 38).*

```
SELECT New.ID
```

```
FROM (SELECT ID, MAPPING(Position, Time), T.Region
          FROM Cyclone, Island, TABLE(decompose(Extent)) AS T
          WHERE Name = "Maui") AS New (ID, Trajectory, Tstart, Tend)
GROUP BY New.ID
HAVING CONTAIN(New.Region, New.Trajectory)
```

Decompose(Extent) is a table function which decompose a polygon (Extent) into a set of triangles (Region). The triangulation algorithm can be found in [73, 62].

5.5 Future Work

Indexing Spatio-temporal databases manage data whose geometry changes over time. While there has been a large amount of research for indexing temporal and spatial data, indexing spatio-temporal data is an area still under research.

One way to index spatio-temporal data is to treat the time axis as just another dimension like the spatial ones and then a traditional spatial access method such as the R-trees [38] or Quadtree [70]. These methods approximate spatial objects by their *Minimum Bounding Rectangles* (MBRs) in order to construct the spatial index, which is inefficient solution if the time span of the spatio-temporal object is long or if the object's geometry changes rapidly.

[89] discusses the following issues of efficiently indexing spatio-temporal objects:

- data types and data sets supported — should support spatial access methods and at least one dimension of time (valid time or transaction time).
- index construction — should support not only bulk loading but also dynamic insertion/update, and change of timestamp granularity should be allowed.
- query processing — should support fundamental query types such as selection and join queries as well as some specialized queries such as "nearest neighbor" queries.

A data structure suitable for storing and retrieving time intervals and directed triangles, and meets the above mentioned criterion will need to be created.

Another issue worth considering is how to index a continuously moving object, especially the ones whose shape change as well as their position.

Spatial Join Spatial joins are one of the most important operations for combining spatial objects stored in several relations. Processing efficient spatial joins is extremely important since its execution time is super-linear in the number of spatial objects of the participating relations and this number may be very large.

There are three types of spatial joins [8]. The first one is *MBR-spatial-join*, which computes all pairs of identifier to each spatial object whose intersection of minimum bounding rectangles of the two spatial objects is not empty; the second one is *ID-spatial-join*, which computes all pairs of IDs with intersection of the two spatial object is not empty; and the last one is *Object-spatial-join*, which computes the intersection of two objects.

Almost all methods designed for an efficient join processing of non-spatial relations cannot be used for spatial joins. Using the simple nested loop approach, every object of one relation has to be checked against all objects of the other relation, thus the performance of the nested loop algorithm is not good. Hashed-based join algorithms are suitable only for natural and equi-joins, but not for spatial joins since some spatial join does not only compute the identifiers of the objects in the response set, but also the resulting objects. An approach similar to sort-merge join may be considered if spatial objects are sorted according to their spatial proximity.

[8] presents techniques for improving spatial join's execution time with respect to both CPU and I/O time using R-trees [38], and in particular R*-trees [5]. It empasizes on computing the spatial join of the minimum bounding rectangles of the spatial objects.

Given the internal spatial data type as directed triangles, it is an interest-

ing direction for future research to exploit spatial access method for an efficient join processing. Using parallel computer systems and disk arrays will also be interesting for performing spatial joins.

5.6 Summary

This chapter proposes an extendable data model and query language for spatio-temporal information. SQL^{ST} maintains orthogonality of temporal and spatial aspects of data, and has minimal additions to SQL. In fact, only user-defined aggregates (UDAs) are needed to accomplish the spatio-temporal queries. Moreover, end-users can have their own set of spatio-temporal operators implemented as UDAs, and they can also choose which level of abstraction they want to work on. Future work should focus on better spatial indexes such as R-trees [38] and on spatial join mechanisms.

CHAPTER 6

Conclusions

In this dissertation, we have investigated data models and query languages for spatio-temporal information.

Therefore, we first propose a minimalist approach to represent temporal information using a point-based representation. In addition to standard SQL aggregates, we add user-defined temporal aggregates to boost users' convenience and implementation efficiency. We demonstrate the power and generality of this approach by showing that it can express all valid-time TSQL2 queries, and it can be extended uniformly to SQL, QBE and Datalog.

Next, we describe the TENORS system, which is an implementation of SQL^T. TENORS uses intervals to represent time at the physical level. Temporal constructs are implemented by user-defined functions and table expressions. A usefulness-based storage organization is used in TENORS for performance, scalability, and automatic generation of temporal indices.

Then, we review the properties of spatial objects and introduce *counterclock-*

wise directed triangles as our major abstract spatial data type, which is used together with points and lines to model spatial objects. Algorithms for polygon triangulation, relationships between spatial objects, and spatial operations on the objects are then presented.

Finally, we propose an extendable data model and query language for spatio-temporal information—SQLST. The SQLST system maintains orthogonality of temporal and spatial aspects of data, and has minimal additions to SQL. We employ user-defined aggregates (UDAs) to support spatio-temporal queries; by means of UDAs, end-users can add their own set of spatio-temporal operators. We also provide a more abstract data model for spatio-temporal information, so end-users can choose which level of abstraction they want to work on.

Future and on-going work includes user-defined indices that simplify the mapping between the different levels of representation, supporting better spatial indices such as R-trees [38], and optimizing spatial join mechanisms to expedite the queries.

In summary, our approach provides better data models and query languages for temporal and spatio-temporal information in terms of usability, generality, flexibility, extensibility, and compatibility with Object-Relational systems. The multi-level architecture lets end-users choose the level of abstraction that they want to work on. The use of user-defined aggregates makes extensions easier

to implement on Object-Relational systems. Moreover, further extensions and customization by end-users are also supported via user-defined spatio-temporal aggregates. The performance results we have obtained are also quite encouraging.

References

[1] S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases*, Addison-Wesley, 1995

[2] J.F. Allen. Maintaining knowledge about temporal intervals. In *Communications of the ACM*, Vol.26, No.11, pp.832-843, 1983

[3] W.G. Aref and H. Samet. An Approach to Information Management in Geographical Applications. In *Proceedings of the 2nd International Symposium on Spatial Data Handling*, pp.589-598, 1990

[4] ARCSS Data and Information Archive, Cyclone Track Data Set, *http://arcss.colorado.edu/Catalog/arcss003.html*

[5] N. Bechmann, H.P. Kriegel, R. Schneider and B. Seeger. The R*–tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 322-331, 1990

[6] M.H. Bohlen, R.T. Snodgrass and M.D. Soo. Coalescing in Temporal Databases. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 180-191, 1996

[7] M.H. Bohlen, R. Busatto and C.S. Jensen. Point-Versus Interval-based Temporal Data Models. In *Proceedings of the 14th International Conference on Data Engineering*, pp.192-200, 1998

[8] T. Brinkhoff, H.P. Kriegel and B. Seeger. Efficient Processing of Spatial Joins Using R–Trees. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp.237-246, 1993

[9] Y.S. Bugrov. *Fundamentals of Linear Algebra and Analytical Geometry*, Moscow : Mir Publishers, 1982

[10] T. Burns, et al. Reference Model for DBMS Standardization, Database Architecture Framework Task Group (DAFTG) of the ANSI/X3/SPARC Database System Study Group. In *SIGMOD Record* Vol.15, No.1, pp.19-58, 1986

[11] M. Cai, D. Keshwani and P.Z. Revesz. Parametric Rectangles: A Model for Querying and Animation of Spatiotemporal Databases. In *Proceedings of the 7th International Conference on Extending Database Technology*, pp.430-444, 2000

[12] B. Chazelle. A Theorem on Polygon Cutting with Applications. In *Proceedings of the 23rd IEEE Symposium on Foundation of Computer Science*, pp.339-349, 1982

[13] D. Chamberlin. *A complete Guide to DB2 Universal Database*, Morgan Kaufmann, 1998

[14] B. Chazelle and J. Incerpi. Triangulation and Shape Complexity. In *ACM Transactions on Graphics*, Vol.3, No.2, pp.135-152, 1984

[15] C.X. Chen and C. Zaniolo. Universal Temporal Data Languages. In *Proceedings of the 6th International Workshop on Deductive Databases and Logic Programming*, 1998

[16] C.X. Chen and C. Zaniolo. Universal Temporal Extensions for Data Languages. In *Proceedings of the 15th International Conference on Data Engineering*, pp.428-437, 1999

[17] C.X. Chen and C. Zaniolo. SQLST: A Spatio-Temporal Data Model and Query Language. In *Proceedings of the 19th International Conference on Conceptual Modeling*, pp.96-111, 2000

[18] J. Chomicki and P.Z. Revesz. Constraint-Based Interoperability of Spatiotemporal Databases. In *Advances in Spatial Databases*, LNCS 1262, pp.142-161, Springer, 1997

[19] J. Chomicki and P.Z. Revesz. A Geometric Framework for Specifying Spatiotemporal Objects. In *Proceedings of the 6th International Workshop on Time Representation and Reasoning*, pages 31-46, 1999

[20] W.W. Chu, C-C. Hsu, A.F. Cardenas and R.K. Taira. Knowledge-Based Image Retrieval with Spatial and Temporal Constructs. In *IEEE Transactions on Knowledge and Data Engineering*, Vol.10, No.6, pp.872-888, 1998

[21] K.L. Clarkon, R.E. Tarjan and C.J. Van Wyk. A Fast Las Vegas Algorithm for Triangulating a Simple Polygon. In *Discrete & Computational Geometry*, Vol.4, No.5, pp.423-432, 1989

[22] K.L. Clarkson, R. Cole and R.E. Tarjan. Randomized Parallel Algorithms for Trapezoidal Diagrams. In *Proceedings of the 7th ACM Symposium on Computational Geometry*, pp.152-161, 1991

[23] P. Dadam, V.Y. Lum and H.D. Werner. Integration of Time Versions into a Relational Database System. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 509-522, 1984

[24] M.J. Egenhofer, A.U. Frank and J.P. Jackson. A Topological Data Model for Spatial Databases. In *Proceedings of Symposium on the Design and Implementation of Large Spatial Databases*, pp.271-286, 1989

[25] M.J. Egenhofer. Spatial SQL: A Query and Presentation Language. In *IEEE Transactions on Knowledge and Data Engineering*, Vol.6. No.1. pp.86-95, 1994

[26] R. Elmasri, G.T.J. Wuu and Y.J. Kim. The Time Index: An Access Structure for Temporal Data. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pp.1-12, 1990

[27] M. Erwig, R.H. Guting, M.M. Schneider and M. Vazirgiannis. Abstract and Discrete Modeling of Spatio-Temporal Data Types. In *Proceedings of ACM International Symposium on Geographic Information Systems*, pp.131-136, 1998

[28] L. Forlizzi, R.H. Guting, E. Nardelli and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp.319-330, 2000

[29] A. Fournier and D.Y. Montuno. Triangulating simple polygons and equivalent problems. In *ACM Transactions on Graphics*, Vol.3, No.2, pp.153-174, 1984

[30] M.R. Garey, D.S. Johnson, F.P. Preparata, and R.E. Tarjan. Triangulating a Simple Polygon. In *Information Processing Letters*, Vol.7, No.4, pp.175-179, 1978

[31] C.H. Goh, H. Lu, B.C. Ooi and K.L. Tan. Indexing Temporal Data Using Existing B+–Trees. In *IEEE Transactions on Knowledge and Data Engineering*, Vol.18, No.2, pp.147-165, 1996

[32] R. Gonzalez and R. Woods. *Digital Image Processing*, Addison-Wesley, 1998

[33] S. Grumbach, P. Rigaux and L. Segoufin. Spatio-Temporal Data Handling with Constraints. In *Proceedings of ACM International Symposium on Geographic Information Systems*, pp.106-111, 1998

[34] S. Grumbach, P. Rigaux and L. Segoufin. The DEDALE System for Complex Spatial Queries. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp.213-224, 1998

[35] R.H. Guting. An Introduction to Spatial Database Systems. In VLDB Journal, Vol.3, No.4, pp.357-399, 1994

[36] R.H. Guting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. In *VLDB Journal*, Vol.4, No.2, pages 243-286, 1995

[37] R.H. Guting, M.H. Bohlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. To appear in *ACM Transactions on Database Systems*

[38] A. Guttman. R–Trees: A Dynamic Index Structure For Spatial Searching. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp.47-57, 1984

[39] R.H. Guting. Geo-Relational Algebra: A Model and Query Language or Geometric Database Systems. In *Proceedings of the 1st International Conference on Extending Database Technology*, pp.506-527, 1988

[40] M. Gyssens, J. Ven den Bussche and D. Ven Gucht. Complete Geometrical Query Languages. In *ACM Symposium on Principles of Database Systems*, pp.62-67, 1997

[41] J.M. Hellerstein, P.J. Haas and H. Wang. Online Aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp.171-182, 1997

[42] IBM DB2 Spatial Extender, *http://www-4.ibm.com/software/data/spatial/*

[43] ESRI Spatial Data Engine, *http://www.esri.com*

[44] Informix Spatial DataBlade Module, *http://spatial.informix.com/module_informix_spatial.html*

[45] C.S. Jensen, et al. A Consensus Glossary of Temporal Database Concepts. In *SIGMOD Record*, Vol.23, No.1, pp.52-64, 1994

[46] C. S. Jensen and R. T Snodgrass. Temporal Data Management. In *IEEE Transactions on Knowledge and Data Engineering*, Vol.11, No.1, pp.36-44, 1999

[47] P.C. Kanellakis, G. Kuper and P.Z. Revesz. Constraint Query Languages. In *Journal of Computer and System Sciences*, special issue edited by Y.Sagiv, Vol.51, No.1, pp.26-52, 1995

[48] D.G. Kirkpatrick. Optimal Search in Planar Subdivisons. In *SIAM Journal on Computing*, Vol.12, No.1, pp.28-35, 1983

[49] G. Kollios, D. Gunopulos and V.J. Tsotras. On Indexing Mobile Objects. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp.261-272, 1999

[50] B. Kuijpers, J. Paredaens and L. Vandeurzen. Semantics in Spatial Databases. In *Semantics in Databases*, LNCS 1358, pp.114-135, Springer, 1998

[51] B. Kuijpers and M. Smits. On Expressing Topological Connectivity in Spatial Datalog. In *Constraint Databases and Their Applications*, LNCS 1191, pp.116-133, Springer, 1997

[52] *LDL++* Version 5, *http://www.cs.ucla.edu/ldl*

[53] D.B. Lomet and B. Salzberg. The Performance of a Multiversion Access Method. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp.353-363, 1990

[54] N.A. Lorentzos and Y.G. Mitsopoulos. SQL Extension for Interval Data. In *IEEE Transactions on Knowledge and Data Engineering*, Vol.9, No.3, pp.480-499, 1997

[55] R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems.* Academic Press, 1992

[56] V.Y. Lum, P. Dadam, R. Erbe, J. Gunauer, P. Pistor, G. Walch, H.D. Werner and J. Woodfill. Designing DBMS Support for the Temporal Dimension. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp.115-130, 1984

[57] Y. Manolopoulos, Y. Theodoridis and V.J. Tsotras. *Advanced database indexing.* Kluwer Academic, 2000

[58] B. Moon, I.F.V. Lopez and V. Immanuel. Scalable Algorithms for Large Temporal Aggregation. In *Proceedings of the 16th International Conference on Data Engineering*, pp.145-154, 2000

[59] R. Muntz, E. Shek and C. Zaniolo. Using *LDL++* For Spatio-Temporal Reasoning in Atmospheric Science Databases. In *Applications of Logic Databases (R. Ramakrishnan, eds.)*, pp. 101-119, 1995

[60] M.A. Nascimento and M.H. Dunham. Indexing Valid Time Databases via B+–Trees. In *IEEE Transactions on Knowledge and Data Engineering*, Vol.11, No.6, pp.929-947, 1999

[61] Oracle Spatial, *http://otn.oracle.com/products/spatial/*

[62] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1998

[63] G. Ozsoyoglu and R.T. Snodgrass. Temporal and Real-Time Databases: A Survey. In *IEEE Transactions on Knowledge and Data Engineering*, Vol.7, No.4, pp.513-532, 1995

[64] J. Paredaens. Spatial Databases, The Final Frontier. *Proceedings of the 5th International Conference on Database Theory*, pp.14-32, 1995

[65] J. Patel, *et al.* Building a Scalable Geo-Spatial DBMS: Technology, Implementation and Evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 336-347, 1997

[66] J. Paredaens and B. Kuijpers. Data Models and Query Languages for Spatial Databases. In *Data & Knowledge Engineering*, Vol.25, No.1-2, pp.29-53, 1998

[67] F. Preparata and M. Shamos. *Computational Geometry: An Introduction.* Springer, 1988

[68] J. Paredaens, J. Van den Bussche and D. Van Gucht. Towards a Theory of Spatial Database Queries. In *ACM Symposium on Principles of Database Systems*, pp.279-288, 1994

[69] R. Ramakrishnan. *Database Management Systems*, WCB/McGraw-Hill, 1998

[70] H. Samet. The Quadtree and Related Hierarchical Data Structures. In *ACM Computing Surveys*, Vol.16, No.2, pp.187-260, 1984

[71] H. Samet. *The design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990

[72] M. Schneider. *Spatial Data Types for Database Systems*, LNCS 1288. Springer, 1997

[73] R. Seidel. A Simple and Fast Incremental Randomized Algorithm For Computing Trapezoidal Decompositions and For Triangulating Polygons. In *Computational Geometry: Theory and Applications*, Vol.1, No.1, pp.51-64, 1991

[74] D. Son and R. Elmasri. Efficient Temporal Join Processing Using Time Index. In *Proceedings of the 8th International Conference on Scientific and Statistical Database Management*, pp.252-261, 1996

[75] A. Segev and H. Gunadhi. Event-Join Optimization in Temporal Relational Databases. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pp.205-215, 1989

[76] T. Sellis, N. Roussopoulos and C. Faloutsos. The R+–Tree: A Dynamic Index For Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pp.507-518, 1987

[77] S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, X. Liu, and C.T. Lu. Spatial Databases - Accomplishments and Research Needs. In *IEEE Transactions on Knowledge and Data Engineering*, Vol.11, No.1, pp.45-55, 1999

[78] A.P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. Modeling and Querying Moving Objects. In *Proceedings of the 13th International Conference on Data Engineering*, pp.422-432, 1997

[79] Sleepycat Software. The Berkeley Database (Berkeley DB). *http://www.sleepycat.com*

[80] R.T. Snodgrass. The Temporal Query Language TQuel. In *Proceedings of the 3rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp.204-213, 1984

[81] R.T. Snodgrass, I. Ahn, G. Ariav, D. Batory, et al. A TSQL2 Tutorial. In *SIGMOD Record*, Vol.23, No.3, pp.27-33, 1994

[82] R.T. Snodgrass, et al. *The TSQL2 Temporal Query Language*, Kluwer Academic, 1995

[83] SQL92. "Information technology – Database Language – SQL." *ISO/IEC 9075:1992.*

[84] SQL99. "Information technology – Database languages – SQL." *ISO/IEC 9075:1999.*

[85] M. Stonebraker, J. Frew, K. Gardels and J. Meredith. The SEQUOIA 2000 Storage Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp.2-11, 1993

[86] M. Stonebraker, P. Brown and D. Moore. *Object-relational DBMSs: tracking the next great wave*, Morgan Kaufmann, 1999

[87] A. Tansel, et al. *Temporal Databases: Theory, Design and Implementation*, Benjamin/Cumming, 1993

[88] R.E. Tarjan and C.J. Van Wyk. An $O(n \log \log n)$ -time Algorithm for Triangulating a Simple Polygon. In *SIAM Journal of Computing*, Vol 17, pp.143-178, 1988

[89] Y. Theodoridis, T. Sellis, A. Papadopoulos and Y. Manolopoulos. Specifications for Efficient Indexing in Spatiotemporal Databases. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pp.123-132, 1998

[90] D. Toman. Point vs. Interval-based Query Languages for Temporal Databases. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp.58-67, 1996

[91] D. Toman. A Point-Based Temporal Extension of SQL. In *Proceedings of the 6th International Conference on Deductive and Object-Oriented Databases*, pp.103-121, 1997

[92] D. Toman. Point-Based Temporal Extensions of SQL and their Efficient Implementation. In *Temporal Databases: Research and Practice (O. Etzion, et al., eds.)*, Springer-Verlag, pp.211-237, 1998

[93] V.J. Tsotras, C.S. Jensen and R.T. Snodgrass. An Extensible Notation for Spatiotemporal Index Queries. In *SIGMOD Record*, Vol.27, No.1, pp.47-53, 1998

[94] University Information System (UIS) Data Set, *http://www.cs.auc.dk/research/DP/tdb/TimeCenter/*

[95] H. Wang and C. Zaniolo. User Defined Aggregates in Object-Relational Systems. In *Proceedings of the 16th International Conference on Data Engineering*, pp.135-144, 2000

[96] H. Wang and C. Zaniolo. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. In *Proceedings of 26th International Conference on Very Large Data Bases*, pp.166-175, 2000

[97] O. Wolfson, B. Xu, S. Chamberlain and L. Jiang. Moving Objects Databases: Issues and Solutions. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pp.111-122, 1998

[98] M.F. Worboys. A generic model for planar geographical objects. *International Journal of Geographic Information Systems*, Vol.6, No.5, pp.353-372, 1992

[99] M.F. Worboys. A Unified Model for Spatial and Temporal Information. *Computer Journal*, Vol.37, No.1, pp.26-34, 1994

[100] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. Technical report, Stanford University, 2000

[101] J. Yang, H.C. Ying and J. Widom. TIP: A Temporal Extension to Informix. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp.596, 2000

[102] C. Zaniolo. A Short Overview of *LDL++*: A Second-Generation Deductive Database System. In *Computational Logic*, Vol.3, No.1, pp.87-93, 1996

[103] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass and R. Zicari. *Advanced Database Systems*, Morgan Kaufmann, 1997