Literature Review on Temporal, Spatial, and Spatiotemporal Data Models

Seo-Young Noh

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Databases and Information Systems Commons, Systems Architecture Commons, and the Theory and Algorithms Commons

Recommended Citation

http://lib.dr.iastate.edu/cs_techreports/150

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
1 Temporal Query Languages

1.1 Introduction

Conventional database systems are capable of storing only the current perception of reality and the current relationship among objects. Such databases enforce currency of data by excluding old data values when newer ones become available. After such currency updates, the old values are lost from the logical level and only the current state remains available. On the other hand, temporal database systems are capable of storing multiple versions of data, thereby allowing users the facility of examining complete object histories [1]

Chomicki defined a temporal database as a repository of temporal information and a temporal query language as any query language for temporal databases in [2]. Jensen in [3] argued that most applications of database technology are temporal in nature and listed applicable examples such as financial applications, record-keeping applications, scheduling applications, and scientific applications. He also pointed out that temporal database management is a vibrant field of research and an active community of several hundred researchers had produced some 2000 papers 1.

Temporal databases can be classified into two groups based on schemes of timestamping. The first one is a tuple level timestamping and the second one is an attribute level timestamping. Temporal databases also can be classified into three categories based on the feature of timestamps—point-based, interval-based, and temporal element-based. Therefore, there are six different combinations possible. The parametric database is the attribute timestamping and temporal elements are used as timestamp.

In this chapter, we will discuss temporal data models and their temporal query languages. There exist many temporal query languages and examples include TQuel, TSQL2, SQLT, IXSQL, SQL/TP, TOLAP, TOSQL, HSQL, ChronoSQL, SQL/Temporal, and so on. Since ParaSQL is an extended SQL and we need to compare it with other temporal query languages, we only select four temporal query languages—SQL/TP, IXSQL, SQLT, and TSQL2. All queries used in this chapter are from literature describing each query language.

1His thesis was published in April 2000.
1.2 SQL/TP

1.2.1 SQL/TP Data Model

In SQL/TP, point-based references to time is used as the basis of the query language. The domain of time is viewed as a discrete countably infinite linearly ordered set without endpoints (e.g., the integers). The individual element of the set represents the actual time instant while the linear order represents the progression of time [4].

In the comparison with ParaSQL, we use an abstract temporal database with relation INDEPENDENCE defined as follows:

INDEPENDENCE (Name, Year)

The INDEPENDENCE relation captures the independence of countries in central Europe. Figure 1 shows the abstract data for INDEPENDENCE relation.

<table>
<thead>
<tr>
<th>INDEPENDENCE</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Year</td>
<td></td>
</tr>
<tr>
<td>Poland</td>
<td>1025</td>
<td>Czechoslovakia</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>Czechoslovakia</td>
</tr>
<tr>
<td>Poland</td>
<td>1794</td>
<td>...</td>
</tr>
<tr>
<td>Poland</td>
<td>1918</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>Czech Republic</td>
</tr>
<tr>
<td>Poland</td>
<td>1938</td>
<td>...</td>
</tr>
<tr>
<td>Poland</td>
<td>1945</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Czech Kingdom</td>
<td>1198</td>
<td>Slovakia</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>Slovakia</td>
</tr>
<tr>
<td>Czech Kingdom</td>
<td>1620</td>
<td>...</td>
</tr>
<tr>
<td>Czechoslovakia</td>
<td>1918</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Independence countries in central Europe [4]

1.2.2 SQL/TP Syntax

SQL/TP consists of two basic syntactic constructs—Select block and Set operations. Select block is the main block of SQL/TP and it is similar to the standard SQL as shown below:

```
SELECT <list of attribute identifiers>
FROM   <sequence of relations>
[WHERE <conditions>]
[GROUP BY <list of attribute identifiers>]
```

Set operations provide a way to combine the individual select blocks using set operations such as union, except, intersect, and so on.
1.2.3 Examples of Queries

1. List all countries that were independent while Czech Kingdom was independent.

SQL/TP:

```sql
SELECT I1.Name
FROM INDEP I1, INDEP I2
WHERE I2.Name = 'Czech Kingdom'
AND I1.Year = I2.Year
```

2. List all years when no country was independent.

SQL/TP:

```sql
SELECT t AS Year
FROM TRUE
EXCEPT
SELECT Year
FROM INDEP
```

3. List all countries that became independent before Slovakia.

SQL/TP:

```sql
SELECT Name
FROM INDEP, (SELECT min(Year) AS y0
FROM INDEP
WHERE Name='Slovakia')
WHERE Year < y0
```

1.3 IXSQL

1.3.1 IXSQL Data Model

IXSQL differs from all the other temporal query languages in that it does not provide support for a special, built-in notion of time. Rather IXSQL adds the ability to define columns of a parameterized interval abstract data type, and it provides special query facilities for manipulating tables with rows that have such interval values. There exists at least two different versions of IXSQL and we consider the latter version in [5, 6].

IXSQL stands for Internal Extension to SQL and it is syntactically and semantically upwards consistent with SQL2. Time intervals are used to mark the duration of events. For the comparison, we use the following relations. and Figure 2 shows a database for the relations [6].

```
TRANSPLANTATION (Name, Date)
DRUG (Name, Drug, Level, Time)
INFECTION (Name, Cause, Time)
```

---

2We use INDEP instead of INDEPENDENCE.
A date has format ‘yyyy-mm-dd,’ but, for simplicity reason, it is represented by an integers, preceded by ‘d.’ Thus, ‘d30’ could represent 1993-05-01. Similarly, [d30, d40) could represent the interval [1993-05-01, 1993-05-11).

1.3.2 Syntax of IXSQL

Line (1)–(5) are executed as in SQL2 and lines (6)–(8), if they exist, are executed in this order. Line 6 (reformats a table with respect to a sequence of columns): REFORMAT AS introduces a sequence of unfold or fold operations which have to be performed on the table retrieved by the
execution of the code in lines (1)–(5). Since REFORMAT AS is applied to the table obtained by the execution of line (1)–(5), the <reformat column list>, which follows in UNFOLD or FOLD, must be a sublist of the columns in <select list>. In particular, if R is the table obtained by the execution of lines (1)–(5) then FOLD $A_1, A_2, \cdots, A_n$ and UNFOLD $A_1, A_2, \cdots, A_n$ are, respectively equivalent to the algebraic operations $\text{fold}[A_1, A_2, \cdots, A_n](R)$ and $\text{unfold}[A_1, A_2, \cdots, A_n](R)$ [6].

Basically, unfold operation changes the underlying structure of an interval-based relation to a point-based relation, and fold operation is the reverse of unfold operation.

1. Give the level of Cyclosporine administered to John, on each of the dates in [d32, d38).

```
IXSQL:
SELECT intersect(Time, [d32, d38)), Level
FROM DRUG
WHERE Name = 'John' AND Drug='Cyclosporine'
   AND Time cp [d32, d38]
REFORMAT AS UNFOLD 1
```

2. List the days within [d31, d35) on which John was suffering from some disease.

```
IXSQL:
SELECT intersect(Time, [d31, d35))
FROM INFECTION
WHERE Name = 'John' AND Time cp[d31, d35]
REFORMAT AS UNFORLD ALL 1
```

3. Give the patients who were administered with Cyclosporine for all the days in [d30, d41).

```
IXSQL:
SELECT DISTINCT Name
FROM DRUG D1
WHERE [d30, d41] subinterv ANY
   (SELECT Time
    FROM DRUG D2
    WHERE D1.Name = D2.Name
    AND D2.Drug = 'Cyclosporine'
    NORMALIZE ON Time)
```

1.4 SQL$^T$

1.4.1 SQL$^T$ Data Model

The basic approach of SQL$^T$ is based on a point-based temporal data model and on explicit time queries. Data model in SQL$^T$ assumes that the use of some granularity for representing valid time—for instance days, 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. Thus, the valid time has become the last column in SQL$^T$ data model. Let’s consider the following relation provided in [7].

```
PREScript (Name, Physician, Drug, Dosage, Frequency, VTime)
```
Since there are not concrete databases for the relation in [8, 9, 10], let’s assume that there exists an imaginary database for the relation. Things that we have to note here are that the data model used in SQL\textsuperscript{T} is a point-based, a time column is explicit for queries, and timestamps are labeled in the tuple level. Therefore, we can consider that the data model of SQL\textsuperscript{T} is similar to that of SQL/TP. In PRESCRIPT relation, the granularity of VTime is day.

1.4.2 Syntax of SQL\textsuperscript{T}

The syntax of SQL\textsuperscript{T} is very similar to SQL/TP because both use a point-based model for representing time granules. The following syntax shows the subset of SQL\textsuperscript{T} syntax.\textsuperscript{3}

\[
\text{SELECT <attribute list> FROM <relation list>} [\text{WHERE <where condition>}] [\text{<group by clause>}] [\text{<having clause>}]\]

1.4.3 Examples of Queries

1. Find the patients who have been prescribed some drug for more than 240 days.

SQL\textsuperscript{T}:

```
SELECT Name
FROM PRESCRIPT
GROUP BY NAME, DRUG
HAVING LENGTH (Vtime) >240
```

2. Find the patients who have been prescribed, throughout 1996.

SQL\textsuperscript{T}:

```
SELECT P.Name
FROM PRESCRIPT AS P
WHERE ((SELECT P1.Name
FROM PRESCRIPT AS P1
WHERE P1.Name = P.Name
AND P1.Drug = P.Drug
AND P1.Drug = 'Proventil')
CONTAINS (SELECT C.Vtime
FROM Calendar AS C
WHERE C.Year = 1996)
)
GROUP BY P.Name
```

3. Find the patients who have been prescribed some drugs for more than 240 consecutive days.

\textsuperscript{3}I could not find a concrete syntax for the SQL\textsuperscript{T} in [8, 9, 10], but we can construct the essential subset of the language because it is a minimal extension of SQL.
SQLT:

CREATE VIEW PartionedP (Name, Drug, PerNo, VTime) AS
SELECT P1.Name, P1.Drug, COUNT(P2.VTime), P1.VTime
FROM PRESCRIPT AS P1 P2
    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

SELECT Name
FROM Partitioned P
GROUP BY Name, Drug, PerNo
HAVING LENGTH (Vtime) > 240

1.5 TSQL2

1.5.1 TSQL2 Data Model

The TSQL2 [5, 2, 11, 12] specification was developed at the University of Arizona By Richard Snodgrass. TSQL2 data model is based on tuple timestamping and 1NF. Time in TSQL2 is multi-dimensional-(valid time and transaction time, or bitemporal). Valid time concerns the time when a fact is true in reality. Transaction time concerns the time the fact was present in the database as stored data. TSQL2 data model uses implicit timestamps. The transaction time of facts are supported by the system itself. In contrast, the valid times of facts are usually supported by the user. Let’s consider the following relations from [11].

EMPLOYEE(ID, Name, Salary, Gender, D-born, DeptName)
SKILL(EmpID, Name)
DEPARTMENT(DeptName, Budget, MgrID)

Figure 3 shows table instances for EMPLOYEE, SKILLS, and DEPARTMENT relations.

1.5.2 Syntax of TSQL2

The syntax of TSQL2 can be referred in [12]. The following syntax shows a subset of the syntax of TSQL2.

SELECT [SNAPSHOT] <select list>
FROM <table source> [AS <correlation>]
[WHERE <search condition>]
[<group by clause>]
[<having clause>]

In [11], the timestamps are formatted as [mm/dd/yyyy-mm/dd/yyyy]. In order to simplify the relations, we ordered the timestamps and allocated unique numbers to the timestamps. The queries are not affected by the changes.
SNAPSHOT specifies that the resulting table will be a snapshot table. The SNAPSHOT keyword tells the query to return a table without a valid-time column, i.e., a non-temporal table [13].

1.5.3 Examples of Queries

1. Find ED’s salaries when he worked in the same department as DI

TSQL2:

```
SELECT SNAPSHOT E3.Salary
FROM EMP(ID, DeptName) AS E1 E2,
     E1(Salary) AS E3
WHERE E1.ID = 'ED' AND E2.ID = 'DI'
     AND E1.DeptName = E2.DeptName
```

2. Find the names of departments that always had a budget greater than $90K during the times when managed by someone named Di.

TSQL2:

```
SELECT SNAPSHOT D2.Name
FROM DEPT(ID, Name, MgrID) AS D1
     DEPT(ID, Budget) (PERIOD) AS D2
 WHERE D1.MgrID = D2.MgrID
     AND D2.Budget > 90
```

---

\(^5\)In the following examples, we use EMP and DEPT as names of EMPLOYEE and DEPARTMENT, respectively
3. When did ED work in Toy department while the department was managed by DI?

**TSQL2:**

```sql
SELECT SNAPSHOT INTERSECT(VALID(E1), VALID(E2))
FROM EMP(ID, Name, Salary, DeptName) AS E1, E2
DEPT(MgrID, Name) AS D1
WHERE E1.Name = 'Di' AND E2.Name = 'Ed'
    AND E1.ID = D1.MgrID
    AND D1.Name = 'Toy'
    AND E2.DeptName = 'Toy'
    AND VALID(D1) OVERLAPS VALID(E2)
```

### 1.6 Summary

In this chapter, we discussed four different temporal query languages and the query examples provided in the papers describing each language. As we have seen, ParaSQL can express the queries in easy and natural ways.

Temporal data model introduced in this chapter can be classified into two groups—implicit time data model and explicit time data model based on approaches to access time granules. Therefore, TSQL2 is in the implicit time data model and the others in explicit time data model. Because of the implicit time, TSQL2 time columns cannot be explicitly referred in the SELECT and WHERE clauses of an SQL query [9].

The temporal query languages except ParaSQL also can be classified into two categories—point-based models and interval-based models. SQL and SQL/TP use point-based model and TSQL2 use interval-based model. IXSQL can change the underlying interval-based structure to point-based structure by calling `unfold` function.

Because of the drawbacks of interval-based and time implicit model, there is no implementation of TSQL2 even though TSQL2 is the standard for temporal query languages [13].

No matter what models—either a point-based model or an interval-based model—the temporal query languages are built on, they cannot satisfy the closure properties on union, intersection, and complementation. That is the main reason that the queries expressed by SQL/TP, IXSQL, SQL, and TSQL2 are so complicated compared to those by ParaSQL.

### 2 Spatial Query Languages

#### 2.1 Introduction

The purpose of spatial databases is to correlate data in space and they provides answers to questions such as how far has waste product extended from the spill location? How many miles away is the closest hospital of his house? Most spatial databases do not stand on their own, but instead are just

---

*Footnote: In [8], TSQL2 is classified as an interval-based model, but the authors in [5] argue that TSQL2 is classified as a point-based model.*
an extension to relational databases. They use a dialect of SQL called Spatial Feature Structured Query Language—which simply adds spatial functions to SQL—such as distance, touches, centroid, inside, area, and extent [14]. Table 1 shows the usability of Spatial Database Management Systems (SDBMS).

<table>
<thead>
<tr>
<th>User</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile phone user</td>
<td>Where is the nearest gas station?</td>
</tr>
<tr>
<td>Army field commander</td>
<td>Has there been any significant enemy troop movement since last night?</td>
</tr>
<tr>
<td>Medical doctor</td>
<td>Based on this patient’s MRI, have we treated somebody with similar condition?</td>
</tr>
<tr>
<td>Farmer</td>
<td>How can I minimize the use of pesticide on my farm?</td>
</tr>
<tr>
<td>Emergency service</td>
<td>Where is the person calling for help located?</td>
</tr>
<tr>
<td>Transport specialist</td>
<td>How should the road network be expanded to minimize traffic congestion?</td>
</tr>
</tbody>
</table>

At the time of writing, there are many query languages for spatial database management systems found in literature. They can be classified into two categories—extended SQL style query languages and visual spatial query languages. Extended SQL style query languages are SQL/OGIS [15, 16, 17], PSQL [18], QL/G [19], Spatial SQL [20], GeoSQL [21], CSQL [22], GEOQL [23], SQL/SDA [24] and so on. Visual spatial query languages are Spatial-Query-by-Sketch [25, 26], Query-by-Visual-Example [27], Cigales [28], and so on.

Authors in [29] argue that the extended SQL approach is the more natural interface language to query a spatial database since the SQL query language is now widely accepted to query a relational database. Therefore, in this chapter, we will discuss four different extended SQL style spatial query languages—SQL/OGIS, QL/G, SQL/SDA, and PSQL, and compare them with ParaSQL.

## 2.2 SQL/OGIS

### 2.2.1 SQL/OGIS Data Model

The OGIS\(^7\) consortium was formed by major software vendors to formulate an industry wide standard related to GIS interoperability. The OGIS spatial data model can be embedded in SQL3.

The OGIS Consortium [15] has standardized spatial feature geometry and spatial operations. The OGIS specification defines a standard for SQL which supports the storage and query of spatial data. The spatial data is based on the OGIS Geometry Object Model in [15]. The non-instantiable class Geometry serves as the base class with subclasses for Point, Curve(line) and Surface(Polygon).

Conceptually, spatial entities are stored as tables with geometry valued columns. Instances of the entities are stored as rows in a table. Datatypes of spatial attributes are drawn from the Geometry Model while those of non-spatial attributes are from SQL3. Implementation of a spatially-enabled table called the feature table, are described for two target environments: SQL3 and SQL3 with Geometry Types. In the SQL3 environment, a geometry-valued column is implemented.

---

\(^7\)OGIS stands for Open Geodata Interchange Standard.
as a Foreign Key reference into a geometry table. A geometric value is stored using one or more rows in the geometry table. The geometry table may be implemented using either standard SQL numeric types or SQL binary types. In SQL3 with Geometry Types, a geometry-valued column is implemented as a column whose SQL type is drawn from the set of Geometry Types.

The SQL functions (methods) specified by the OGIS specification fall into three categories: 1) basic functions on the Geometry datatypes, 2) operators for testing topological relationships, and 3) functions that support spatial analysis.8

We define COUNTRY, CITY, and RIVER relations in SQL/OGIS data model as follows:

\[
\begin{align*}
\text{COUNTRY} & : (\text{Name, Cont, Pop, GDP, Life-Exp, Shape: Polygon}) \\
\text{CITY} & : (\text{Name, Country, Pop, Captital, Shape: Point}) \\
\text{RIVER} & : (\text{Name, Origin, Length, Shape: LineString})
\end{align*}
\]

The primary keys for each relation scheme are Name attribute in each relation and only Shape attributes are indicated with OGIS data types. Figure 4 shows the tables of the World database.

2.2.2 Syntax of SQL/OGIS

The syntax of SQL/OGIS follows the SQL. OGIS provides Geometry Types and functions used in SQL3. Geometry types includes Point, Curve, Surface and Geometry Collection. Each geometric object is associated with a Spatial Reference System, which describes the coordinate space in which the geometric object is defined. Table 2 shows the functions used in SQL/OGIS.

2.2.3 Examples of Queries

1. Find the names of all countries which are neighbors of the United States in the COUNTRY table.

SQL/OGIS:

\[
\begin{align*}
\text{SELECT} & \quad C1.\text{Name AS ‘Neighbors of USA’} \\
\text{FROM} & \quad \text{COUNTRY} \ C1, \text{COUNTRY} \ C2 \\
\text{WHERE} & \quad \text{Touch(C1.Shape, C2.Shape) = 1} \\
& \quad \text{AND C2.Name = ‘USA’}
\end{align*}
\]

2. The St. Lawrence River can supply water to cities that are within 300 km. List the cities that can use water from the St. Lawrence.

SQL/OGIS:

\[
\begin{align*}
\text{SELECT} & \quad C.\text{Name} \\
\text{FROM} & \quad \text{City} \ C, \text{River} \ R \\
\text{WHERE} & \quad \text{Overlap(C.Shape, Buffer(R.Shape, 300) = 1} \\
& \quad \text{AND R.Name=‘St. Lawrence’}
\end{align*}
\]

3. List the length of the rivers in each of the countries they pass through.

SQL/OGIS:

\[
\begin{align*}
\text{SELECT} & \quad R.\text{Name, C.Name,} \\
& \quad \text{Length(Intersection(R.Shape, C.Shape) AS Length} \\
\text{FROM} & \quad \text{RIVER} \ R, \text{COUNTRY} \ C \\
\text{WHERE} & \quad \text{Cross(R.Shape, C.Shape) = 1}
\end{align*}
\]

---

8For more detail information, refer to [15, 17].
### COUNTRY

<table>
<thead>
<tr>
<th>Name</th>
<th>Cont</th>
<th>Pop(millions)</th>
<th>GDP(billions)</th>
<th>Life-Exp</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canada</td>
<td>NAM</td>
<td>30.1</td>
<td>658.0</td>
<td>77.08</td>
<td>Polygonid-1</td>
</tr>
<tr>
<td>Mexico</td>
<td>NAM</td>
<td>107.5</td>
<td>694.3</td>
<td>69.36</td>
<td>Polygonid-2</td>
</tr>
<tr>
<td>Brazil</td>
<td>SAM</td>
<td>183.3</td>
<td>1004.0</td>
<td>65.60</td>
<td>Polygonid-3</td>
</tr>
<tr>
<td>Cuba</td>
<td>NAM</td>
<td>11.7</td>
<td>16.9</td>
<td>75.95</td>
<td>Polygonid-4</td>
</tr>
<tr>
<td>USA</td>
<td>NAM</td>
<td>270.0</td>
<td>8003.0</td>
<td>75.75</td>
<td>Polygonid-5</td>
</tr>
<tr>
<td>Argentina</td>
<td>SAM</td>
<td>36.3</td>
<td>348.2</td>
<td>70.75</td>
<td>Polygonid-6</td>
</tr>
</tbody>
</table>

### CITY

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Pop(millions)</th>
<th>Capital</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Havana</td>
<td>Cuba</td>
<td>2.1</td>
<td>Y</td>
<td>Pointid-1</td>
</tr>
<tr>
<td>Washington, D.C.</td>
<td>USA</td>
<td>3.2</td>
<td>Y</td>
<td>Pointid-2</td>
</tr>
<tr>
<td>Monterrey</td>
<td>Mexico</td>
<td>2.0</td>
<td>N</td>
<td>Pointid-3</td>
</tr>
<tr>
<td>Toronto</td>
<td>Canada</td>
<td>3.4</td>
<td>N</td>
<td>Pointid-4</td>
</tr>
<tr>
<td>Brasilia</td>
<td>Brazil</td>
<td>1.5</td>
<td>Y</td>
<td>Pointid-5</td>
</tr>
<tr>
<td>Rosario</td>
<td>Argentina</td>
<td>1.1</td>
<td>N</td>
<td>Pointid-6</td>
</tr>
<tr>
<td>Ottawa</td>
<td>Canada</td>
<td>0.8</td>
<td>Y</td>
<td>Pointid-7</td>
</tr>
<tr>
<td>Mexico City</td>
<td>Mexico</td>
<td>14.1</td>
<td>Y</td>
<td>Pointid-8</td>
</tr>
<tr>
<td>Buenos Aires</td>
<td>Argentina</td>
<td>10.75</td>
<td>Y</td>
<td>Pointid-9</td>
</tr>
</tbody>
</table>

### RIVER

<table>
<thead>
<tr>
<th>Name</th>
<th>Origin</th>
<th>Length(kilometers)</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rio Parana</td>
<td>Brazil</td>
<td>2600</td>
<td>LineStringid-1</td>
</tr>
<tr>
<td>St. Lawrence</td>
<td>USA</td>
<td>1200</td>
<td>LineStringid-2</td>
</tr>
<tr>
<td>Rio Grande</td>
<td>USA</td>
<td>3000</td>
<td>LineStringid-3</td>
</tr>
<tr>
<td>Mississippi</td>
<td>USA</td>
<td>6000</td>
<td>LineStringid-4</td>
</tr>
</tbody>
</table>

Figure 4: The World database in SQL/OGIS data model [17]

### 2.3 QL/G

#### 2.3.1 QL/G Data Model

QL/G stands for Query Language for Geometric databases. It has been developed at the University of Waterloo and is intended to be a general-purpose spatial query language for manipulating both alphanumerical as well as geometric data. QL/G is a modular, strongly-typed functional language with an SQL flavor. QL/G is an extension of SQL and the data model of QL/G is a nested relational model extended with geometric data and operators. However, the geometric data and operators are not dependent on the nested relational model. The geometric component is independent of any existing set-oriented data models, and it takes a modular and functional approach to the design of the query language to achieve the objective. The basic constructs or operators in QL/G can be considered as functions. A function takes in arguments and yields some output. A query is a
<table>
<thead>
<tr>
<th>Function Type</th>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Functions</td>
<td>SpatialReference()</td>
<td>Returns the Reference System of the geometry</td>
</tr>
<tr>
<td></td>
<td>Envelope()</td>
<td>The minimum bounding rectangle of the geometry</td>
</tr>
<tr>
<td></td>
<td>Export()</td>
<td>Convert the geometry into a different representation.</td>
</tr>
<tr>
<td></td>
<td>IsEmpty()</td>
<td>Tests if the geometry is a empty set or not.</td>
</tr>
<tr>
<td></td>
<td>IsSimple()</td>
<td>Returns TRUE if the geometry is simple.</td>
</tr>
<tr>
<td></td>
<td>Boundary()</td>
<td>Returns the boundary of the geometry.</td>
</tr>
<tr>
<td>Topological/</td>
<td>Equal</td>
<td>Tests if the geometries are spatially equal.</td>
</tr>
<tr>
<td>Set Operators</td>
<td>Disjoint</td>
<td>Tests if neither interiors nor boundaries intersect.</td>
</tr>
<tr>
<td></td>
<td>Intersect</td>
<td>Tests if the geometries intersect.</td>
</tr>
<tr>
<td></td>
<td>Touch</td>
<td>Tests if the boundaries intersects and interior does not.</td>
</tr>
<tr>
<td></td>
<td>Cross</td>
<td>Tests if the geometries cross each other.</td>
</tr>
<tr>
<td></td>
<td>Within</td>
<td>Tests if the given geometry is within another given geometry.</td>
</tr>
<tr>
<td></td>
<td>Contains</td>
<td>Tests if the given geometry contains another given geometry.</td>
</tr>
<tr>
<td></td>
<td>Overlap</td>
<td>Tests if the interiors intersect.</td>
</tr>
<tr>
<td>Spatial Analysis</td>
<td>Distance</td>
<td>Returns the shortest distance between two geometries.</td>
</tr>
<tr>
<td></td>
<td>Buffer</td>
<td>Returns a geometry that represents all points whose distance from the given is less than or equal to the specified distance.</td>
</tr>
<tr>
<td></td>
<td>ConvexHull</td>
<td>Returns the convex hull of the geometry.</td>
</tr>
<tr>
<td></td>
<td>Intersection</td>
<td>Returns the intersection of two geometries.</td>
</tr>
<tr>
<td></td>
<td>Union</td>
<td>Returns the union of two geometries.</td>
</tr>
<tr>
<td></td>
<td>Difference</td>
<td>Returns the difference of two geometries.</td>
</tr>
<tr>
<td></td>
<td>SymDiff</td>
<td>Returns the symmetric difference of two geometries.</td>
</tr>
</tbody>
</table>

function whose arguments may be other queries/functions. This feature allows queries be combined in any manner to produce more complicated queries [19].

In QL/G, there are six disjoint sets as shown in Table 3. The set $R \cup S \cup \{\text{TRUE, FALSE} \}$ is said to be the set of *atomic values*. The tokens REAL, STR and BOOLEAN denote the *atomic type*. The tokens POINT, S_LINE, S_REGION, LINE, LINE*, REGION, REGION* are the *geometric data types*. REGION* is of type either REGION, LINE or POINT. Similarly, LINE* denotes either type of LINE or POINT.

### 2.3.2 Syntax of QL/G

QL/G has two main categories—*structure transformation operators* and *geometric operators*. Structure transformation operators are an extension of constructs in SQL to a nested relational model. Geometric operators are defined independently from the nested relational operators, and any other suitable set of operators that produce set values could replace those in structure transformation operators. The language includes algebraic operators like *union* and *minus* as well as all SQL built-in functions such as *distinct, min, max, count, sum* and *average*. However, the operators union, minus and distinct are extended to accept geometric values.
Table 3: Six disjoint set in QL/G

<table>
<thead>
<tr>
<th>Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>the set of real numbers</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>the set of ${\text{TRUE, FALSE}}$</td>
</tr>
<tr>
<td>$B$</td>
<td>${\text{REAL, STR, BOOLEAN, POINT, S_LINE, POINT, S_LINE, S_REGION, LINE, LINE*, REGION, REGION*}}$</td>
</tr>
<tr>
<td>$A$</td>
<td>a countably infinite set of symbols which are called attributes.</td>
</tr>
<tr>
<td>$C$</td>
<td>a countably infinite set of symbols which are called relation names.</td>
</tr>
</tbody>
</table>

SELECT tuple($\text{newname}_1 : \text{result}_1(x_1, \cdots, x_n)$, 
\[ \cdots \]
$\text{newname}_k : \text{result}_k(x_1, \cdots, x_n))$)
FROM $x_1 \text{ IN } \text{expr}_1$, 
\[ \vdots \]
$\text{in } \text{expr}_n$
WHERE qualifications;

Every SELECT-FROM-WHERE block returns a set of tuples defined on $k$ attributes. There are several major differences from the popular SQL. Table 4 shows the description on arguments. For geometric operator, many of the proposed geometric operators are overloaded, meaning that the same name is used for different computations. They includes inside, overlap, adjacent, length, mindist, maxdist, and so on.

Table 4: Description on arguments in QL/G syntax

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{new name}$</td>
<td>New attribute names.</td>
</tr>
<tr>
<td>$\text{qualification}$</td>
<td>The qualification allows nesting of SELECT-FROM-WHERE and is augmented with operators and geometric operators.</td>
</tr>
<tr>
<td>$\text{expr}_i$</td>
<td>Each $\text{expr}_i$ in the FROM clause is evaluated and returns a set of values.</td>
</tr>
<tr>
<td>$x_j$</td>
<td>Each variable $x_j$ is ranging over a value in the corresponding set. The qualification is then evaluated. If it returns TRUE, expressions in the SELECT clause are evaluated as the values of the corresponding columns.</td>
</tr>
<tr>
<td>$\text{result}_i(x_1, \cdots, x_n)$</td>
<td>It is a function returning a value on the current assignment of variables $x_1, \cdots, x_n$.</td>
</tr>
</tbody>
</table>
2.3.3 Examples of Queries

For the comparison, let’s define following relations whose italic attributes form the key of the corresponding relation.

\[
\begin{align*}
\text{CITIES} & \quad (\text{Name:STR, Surface:S_REGION}) \\
\text{HIGHWAYS} & \quad (\text{Name:STR, Route:S_LINE}) \\
\text{ROADS} & \quad (\text{City:STR, Streets(\text{Name:STR, Route:S_LINE})}) \\
\text{RIVERS} & \quad (\text{Name:STR, Route:LINE}) \\
\text{OIL} & \quad (\text{Oid:REAL, Potential: REAL, Surface: REGION}) \\
\text{COAL} & \quad (\text{Cid:REAL, Potential: REAL, Surface: REGION})
\end{align*}
\]

1. Find all neighboring cities of Toronto.

\[
\text{QL/G:} \\
\begin{align*}
\text{SELECT tuple(neighbor: Other.name)} \\
\text{FROM Toronto IN CITIES, Other IN CITIES} \\
\text{WHERE Toronto.Name='Toronto' AND} \text{Other.Surface ADJACENT Toronto.Surface};
\end{align*}
\]

2. Consider a geological exploration application. Based on two preliminary independent geological surveys on oil and coal, a province is partitioned into a large number of regions according to their potentials. The information is recorded in the relations OIL and COAL, respectively. The higher the number of potential, the greater the chance we find oil or coal in a region. Notice that, however, we need to do the actual drilling to verify the existence of either kind of reserves. Find the regions that the high in potential (>80) for both oil and coal.

\[
\text{QL/G:} \\
\begin{align*}
\text{DISTINCT (UNNEST(} \\
\text{SELECT tuple(high_pot:} \\
\text{regions(intersection(c.Surface, o.Surface))))} \\
\text{FROM c IN COAL, o IN OIL} \\
\text{WHERE (c.Potential > 80) AND (o.Potential > 80)} \\
\text{AND (c.Surface OVERLAP o.Surface))} \\
\text{ON high\_pot);}
\end{align*}
\]

3. What are the area of the city Hamilton?\footnote{The original query was to find the area as well as the length of the boundary of the city. To illustrate the difference between two query languages, the query was modified to find only the area of the city.}

\[
\text{QL/G:} \\
\begin{align*}
\text{SELECT tuple(size: area(H.Surface))} \\
\text{FROM H IN CITIES} \\
\text{WHERE H.Name = ‘Hamilton’;}
\end{align*}
\]
2.4 SQL/SDA

2.4.1 SQL/SDA Data Model

SQL/SDA (Spatial Data Analysis) has been designed to satisfy the requirement that GIS development is to provide easy and effective access to spatial analysis functionalities for supporting decision making based on geo-referenced data within the framework of SQL standard for spatial extensions. Within such a framework, the objective of SQL/SDA is to support the expression of complicated spatial queries dealing with various spatial analysis problems. SQL/SDA adopts SQL with geometry types and a geometry-valued column is implemented as a column whose data type is drawn from the set of geometry types which are defined by OGIS [24].

In SQL/SDA data model, we can define LANDUSE, SOIL, BUILDING, SEWER, STREAM, and SHOP relations by using geometry datatypes. Creating tables are exactly following the standard SQL except it uses geometry datatypes. In the tables, attribute Location has one of geometry types defined in SQL/SDA.

- LANDUSE (ID, Type, Location)
- SOIL (ID, Type, Location)
- BUILDING (ID, Name, Owner, Usage, Location)
- SEWER (ID, Type, Capacity, Location)
- STREAM (ID, Name, Location)
- SHOP (ID, Name, Location)

In the above feature table, the LANDUSE, SOIL and BUILDING features are of polygon type, the SEWER and STREAM of linestring type, and the SHOP of point type. Figure 5 shows maps and their corresponding relational tables.

![Figure 5: LANDUSE and SOIL relations in the SQL/SDA data model [24]](image)

2.4.2 Syntax of SQL/SDA

The syntax of SQL/SDA is based on the conventional SQL and only the FROM clause is different from the standard SQL. The basic syntax of SQL/SDA and the BNF form for FROM clause are defined as follows:
The spatially derived attributes are treated in the same way as those in the source relations and, thus, can be applied as constraints in the main WHERE clause, and/or referenced for further analysis, aggregation, or graphical display in the main SELECT clause. Since SQL/SDA needs to comply with both the general spatial analysis procedure and the SQL concepts, the subquery in the FROM clause is employed [24].

2.4.3 Examples of Queries

1. Find the land parcels and their corresponding area on the condition that the landuse type of each parcel is brushland and soil type is ‘A’ and area is between 700 hectares to 900 hectares.

SQL/SDA:

```
SELECT lu.ID, sl.ID, ILocation, areaval
FROM (SELECT *, OVERLAP(lu.Location, sl.Location) AS overlapval,
       INTERSECTION(lu.Location, sl.Location) AS lLocation,
       AREA(lLocation) AS areaval
       FROM LANDUSE AS lu, SOIL AS sl)
WHERE lu.type='Brushland' AND sl.type='A'
AND overlapval=TRUE AND areaval > 700
AND areaval < 900
```

2. This query is to select a lab site. The selection criteria are:

(a) Preferred landuse is brushland.

(b) Soil type should be ‘A.’

(c) Site must be within 300 meters of existing sewer lines.

(d) Site must be beyond 20 meters of existing streams.

(e) Site must contain an area at least 2,000 square meters.
3. The last query concerns site selection. This one is related to land suitability evaluation for building an institute, in which all the possible sites need to be classified into different suitability levels. Assuming that there are two maps: LANDUSE and SOIL, and suitability levels are “high (III),” “medium (II),” and “low (I).” The evaluation includes the following steps:

(a) Overlay the landuse map and soil map.

(b) Classify the overlay map in terms of the evaluation criteria: i) If the landuse type is “Brushland” and soil type is ‘A,’ then the suitability level is ‘III.’ ii) If the landuse type is “Water” and soil type is ‘A,’ then the suitability level is ‘I.’ iii) Otherwise the suitability level is ‘II.’

(c) Merge the parcels (area > 100 hectares) with the same suitability level and display them.

This query is formulated in SQL/SDA as follows:

```sql
SELECT FUSION(ILocation)
FROM (SELECT *
      INTERSECT (lu.Location, s1.Location) AS ILocation,
      AREA(ILocation) AS areaval,
      classfyval =
      (CASE lu.Type || s1.Type
       WHEN 'Brushland' AND 'A' THEN 'III'
       WHEN 'Brushland' AND 'B' THEN 'II'
       WHEN 'Water' AND 'A' THEN 'I'
       WHEN 'Water' AND 'B' THEN 'II'
       WHEN 'Forest' AND 'A' THEN 'II'
       WHEN 'Forest' AND 'B' THEN 'II'
       END)
      FROM LANDUSE AS lu, SOIL AS s1)
```
WHERE ILocation <> NULL AND areaval > 100
GROUP BY classfyval

2.5 PSQL

2.5.1 PSQL Data Model

PSQL (Pictorial SQL) extends the definition of relations over spatial and other types of domains. Every domain in PSQL is an abstract data type. PSQL supports three basic pictorial domains: points, line segments, and regions. In addition to these three basic pictorial domains, PSQL also supports the standard alphanumeric domains, integers, reals, and strings.

Relations are defined over alphanumeric and/or pictorial domains. They model inter-domain relationships. Every tuple models a relationship among those alphanumeric and pictorial elements of the domains [18]. The following relation is defined over a set of pictorial domains (point, segment, region).

COUNTY (CName, Crop, CRegion)

The pictorial domain of CRegion is of type “region.” Figure 6 shows COUNTY relation in PSQL data model.

<table>
<thead>
<tr>
<th>COUNTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CName</td>
</tr>
<tr>
<td>story</td>
</tr>
<tr>
<td>story</td>
</tr>
<tr>
<td>orange</td>
</tr>
<tr>
<td>orange</td>
</tr>
<tr>
<td>orange</td>
</tr>
<tr>
<td>polk</td>
</tr>
</tbody>
</table>

Figure 6: COUNTY relation in PSQL data model

2.5.2 Syntax of PSQL

PSQL is based on the SQL and the skeleton of the PSQL is as follows:

```
SELECT <attribute-target-list>
FROM <relation-list>
[ON <picture-list>]
[WHERE <qualification>]
```

The <attribute-target-list> is the set of tuples resulting from the query. The <relation-list> defines the source relations that will be queried. The optional ON <picture-list> in the mapping is a name list that specifies the picture the query is on. This is used when tuples of the same relation
corresponding to different pictures. The ON clause can be omitted provided that the domains of
the relations are associated with a unique picture. The search area on the picture is specified in the
<qualification> clause. It can either be a bound variable or a location given in absolute constant
coordinates or in variable coordinates. The location variable may just be the name of a location
predefined outside the retrieve mapping. Furthermore, a search area in <qualification> may be
followed by a pictorial operator “cover,” “overlap,” etc., followed by another area specification [18].

2.5.3 Examples of Queries

1. Retrieve complete information about counties which grow wheat or corn.

   PSQL:
   ```sql
   SELECT C1.*
   FROM COUNTY C1 C2
   WHERE (C1.Crop = 'wheat' OR C1.Crop = 'corn')
       AND C1.CName = C2.CName
   ```

2. Retrieve complete information about counties which grow wheat and corn.

   PSQL:
   ```sql
   SELECT C1.*
   FROM COUNTY C1 C2 C3
   WHERE (C1.Crop = 'wheat' AND C2.Crop = 'corn')
       AND C1.CName = C2.CName AND C1.CName = C3.CName
   ```

3. Retrieve information about counties that grow wheat.

   PSQL:
   ```sql
   SELECT C1.*
   FROM COUNTY C1 C2
   WHERE C1.Crop = 'wheat'
       AND C1.CName = C2.CName
   ```

4. Retrieve information about counties that do not grow wheat.

   PSQL:
   ```sql
   ( SELECT *
     FROM COUNTY
   )
   DIFFERENCE
   ( SELECT C1.*
     FROM COUNTY C1 C2
     WHERE C1.Crop = 'wheat'
         AND C1.CName = C2.CName
   )
   ```
2.6 Summary

In this chapter, we discussed spatial query languages and compared with ParaSQL. Spatial query languages can be categorized into two classes—SQL style spatial query languages and visual spatial query languages.\textsuperscript{10} It has been argued whether the relational database query language SQL can be successfully extended for spatial applications. However, since SQL is still a popular database language and its functionalities have been enhanced considerably, it is considered as the most preferred option for this study.

For the comparison with ParaSQL, we have looked at SQL style query languages such as SQL/OGIS, a spatial query language for supporting OGIS standard, QL/G, SQL/SDA, and PSQL. As we have seen in the examples excerpted from literature describing the query languages, we could conclude that ParaSQL provides more easy and natural ways to express the queries. In some queries, we have seen that it made quite complicated to transform English queries into the other query languages when changing “and” to “or,” or adding “not” to the English queries. But ParaSQL did not introduce any additional complexity, but it could express the queries in a natural way.

3 Spatiotemporal Query Languages

3.1 Introduction

Many data objects in the real world have attributes about location and time. For example, a database about sea turtles records the location and time data for turtles that carry radio transmitters. Other examples include tracking vehicles with global positioning systems (GPS), mobile phone users within mobile phone networks, and environmental changes over time. Traditional relational database technology is not suitable for managing spatiotemporal data, which are multi-dimensional with complex structures and behaviors \textsuperscript{[30]}.

Spatiotemporal databases have been the focus of considerable research activity over a significant periods. However, there still exist very few prototypes of complete systems, and far less products that provide effective support for applications tracking changes to spatial and aspatial data\textsuperscript{11} over time. This is because the design and implementation of a complete spatiotemporal database is a challenging undertaking, involving extensions to all aspects of a non-spatiotemporal architecture—data model, query language, query optimizer, query evaluator, programming environment, storage manager, indexes, etc \textsuperscript{[31]}.

In the past, research in spatial and temporal data models and database systems has mostly been done independently. Spatial database research has focused on supporting the modeling and querying of geometries associated with objects in a database. Temporal databases have focused on extending the knowledge kept in a database about the current state of the real world to include the past in the two senses of “the past of the real world” (valid time) and “the past state of the database” (transaction time). Nevertheless, many people have felt that the two areas are closely related, since both deal with “dimensions” or “spaces” of some kind, and that an integration field of “spatiotemporal databases” should be studied and would have important applications \textsuperscript{[32]}.

Authors in \textsuperscript{[33]} suggest two directions to accommodate spatial and temporal databases: 1) the embedding of a temporal awareness in spatial systems, and 2) the accommodation of space into temporal data mining systems. And they points out that the former approach has been the more

\textsuperscript{10}In \textsuperscript{[34]}, the authors classify the query languages into three kinds—textual approaches, non-textual approach, and hypermedia approaches.

\textsuperscript{11}Time and spatial independent data, that is, ordinary data over classical databases.
popular because of the relative maturity of geographic information systems and the availability of time-stamped snapshots of geographic/spatial test data.

Erwig in [35] defines spatiotemporal data as a particular example of temporal data in general. For example, we can deal with temporally changing numbers through a type like $\text{time} \rightarrow \text{num}$. Such a temporal number could give the size of the oil spill depending on the time. In [36], the authors define the spatiotemporal databases as set of moving $n$-dimensional figures described by means of an set of tuples $(x_1, x_2, \cdots, x_n; t)$ in $R^n \times R$, where $R$ is the set of real numbers, $(x_1, x_2, \cdots, x_n)$ represent the spatial coordinates of a point in the $n$-dimensional real space $R^n$ and $t$ is the time coordinate in $R$. In [30], the authors define spatiotemporal databases as a database that embodies spatial, temporal, and spatiotemporal database concepts, and captures simultaneously spatial and temporal aspects of data.

Spatiotemporal applications fall into the category of data intensive applications, often referred to as “non-standard”, including, among others, multimedia, VLSI design, and artificial intelligence based systems [37]. Spatiotemporal data analysis plays an important role in many scientific applications like environmental epidemiology and public health. Data analysis of multidimensional data like spatial, temporal and statistical data occurs in many scientific applications and has to be supported by modern database technology. An adequate data model, comfortable querying and special implementation techniques have to be considered [38].

In spite of the fact that there are many different definitions and approaches to spatiotemporal models and query languages in the spatiotemporal literature, the definitions have the common features in that spatiotemporal databases should deal with time and space dimensions and/or the combined dimension. Therefore, query languages for spatiotemporal databases should provide powerful expressive mechanism to express queries asking spatial, temporal, or spatiotemporal object in effective and in natural.

In the following sections, we will discuss four different spatiotemporal query languages–SQL$^{ST}$, E.S. STQL, K.R.P STQL and STSQL. We also compare them with ParaSQL. The all queries have been excerpted from the literature describing each spatiotemporal query language.

3.2 SQL$^{ST}$

3.2.1 SQL$^{ST}$ Data Model

The objective of SQL$^{ST}$ is to minimize the extensions required in SQL to support spatiotemporal queries. SQL$^{ST}$ is based on a directed-triangulation model to represent spatial data, and a point-based model to represent time at the conceptual level. According to [39], the authors defined SQL$^T$ and SQL$^S$ components based on Worboy’s suggestion [40]. Therefore, SQL$^{ST}$ is the combined query language with SQL$^T$ and SQL$^S$. To model time at the conceptual level, a point-based time model is used, where information is repeated for each time granule where it is valid. In the spatial model, SQL$^{ST}$ use triangles to represent polygons; a similar approach was proposed in [41, 42].

The authors explain two reasons for the polygon-oriented representation. The first is that coalescing is needed much less frequently than in temporal queries. The second is that two dimensional shapes offer a more natural representation for many application domains. Therefore, SQL$^{ST}$ views reality as a sequence of snapshots of objects that are moving and/or changing in shape [7, 39].

Figure 7 shows 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 $O2$ moves to a new position. At time $t = 20$, $O1$ has some more changes in shape while $O2$ stays unchanged [39].
An internal representation of Figure 7 could be shown as follows:

\[
\begin{align*}
&(O_1 [(2,6),(2,2),(6,2),(6,6)], [0,10)) \\
&(O_2 [(6,6),(6,2),(10,4)], [0,10)) \\
&(O_1 [(2,6),(2,2),(4,2),(4,4),(6,4),(6,6)], [10,20)) \\
&(O_2 [(4,4),(4,0),(8,2)], [10,20)) \\
&(O_1 [(2,6),(2,2),(6,6)], [20,30)) \\
&(O_2 [(4,4),(4,0),(8,2)], [20,30))
\end{align*}
\]

Here, the regions are represented by a circular list of vertexes, and the time elements are stored as intervals. Figure 8 shows how the changes are recorded in the database at the conceptual level.

<table>
<thead>
<tr>
<th>ID</th>
<th>$x_1$</th>
<th>$y_1$</th>
<th>$x_2$</th>
<th>$y_2$</th>
<th>$x_3$</th>
<th>$y_3$</th>
<th>VTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_1$</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>$O_1$</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>$O_1$</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>$O_1$</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>$O_1$</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>$O_1$</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>$O_2$</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>$O_2$</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>$O_1$</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>$O_1$</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>$x_1$</th>
<th>$y_1$</th>
<th>$x_2$</th>
<th>$y_2$</th>
<th>$x_3$</th>
<th>$y_3$</th>
<th>VTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_1$</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>$O_1$</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>$O_1$</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>$O_1$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>$O_1$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>$O_1$</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>$O_1$</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>29</td>
</tr>
<tr>
<td>$O_1$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>$O_1$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>29</td>
</tr>
</tbody>
</table>

Figure 8: A database for Figure 7 in SQL$^{ST}$ [39]

### 3.2.2 Syntax of SQL$^{ST}$

There are no concrete syntax for SQL$^{ST}$ in [7, 39] at the time of writing. But, we can derive the syntax as shown below because SQL$^{ST}$ is the minimal extension from the standard SQL. It
provides interval operators suggested by Allen [43] such as overlap, precede, contain, equal, meet and intersect. It also support commonly used spatial predicates such as equal, disjoint, overlap, meet, contain, adjacent, and common_border, etc.

3.2.3 Examples of Queries

In this section, we will look at SQL\textsuperscript{ST} queries transforming some examples introduced in [44]. For the comparison with ParaSQL, we consider three relations—FOREST, FOREST\_FIRE, and FIRE\_FIGHTER as shown below:

FOREST (Forestname CHAR(30), Territory REGION, VTime DAY)
FOREST\_FIRE (Firename CHAR(30), Extent REGION, VTime DAY)
FIRE\_FIGHTER (Fightername CHAR(30), Location POINT, VTime DAY)

FOREST relation has records of the location and the development of forests changing over time. FOREST\_FIRE relation has records of the evolution of forest fires. FIRE\_FIGHTER relation has records of the motion of fighters.

When translating a database consisting of those relations into a parametric database, we have to note that each attribute in a relation consists of spatiotemporal dimension and its corresponding value. The primary keys for the relations will be Forestname, Firename, and Fightername, respectively. The columns Territory and Extent have a spatial data type as REGION and Location has a type as POINT; temporal data column VTime has a granularity of DAY in SQL\textsuperscript{ST} data model.

1. When and where did the fire called “The Big Fire” reach what largest extent?

**SQLST:**

```sql
SELECT F1.VTime, F2.extent, AREA(F1.extent)
FROM FOREST\_FIRE as F1 F2
WHERE F1.Firename = ‘The Big Fire’
AND F2.Firename = ‘The Big Fire’
AND F1.VTime = F2.VTime
GROUP BY F1.VTime
HAVING AREA(F1.Extent) = (SELECT MAX(AREA(extent))
FROM FOREST\_FIRE
WHERE Firename = ‘The Big Fire’)
```

**ParaSQL:**

```sql
SELECT *
RESTRICTED TO [
[[PROJ SPACE
FROM [[F.Name=‘The Big Fire’]]
WHERE maxarea(SPACE)]
FROM FOREST\_FIRE F
WHERE F.Firename = ‘The Big Fire’
```
This query is very interesting and makes ParaSQL develop a new sublanguage for internal navigation. The concept of the parametric data model is to store an object in a single tuple, not separately. Therefore, it can retrieve tuples without introducing another variables for combining tuples into one object compared to other spatiotemporal query languages. But it has a limitation to compare parametric elements each other in a same tuple. Comparing parametric elements which exist in a same tuple is one type of restriction of domain of the tuple.

Therefore, we can decide the position of the sublanguage as the inside of RESTRICTED TO clause. Since we handles only spatiotemporal elements, there exist two domains-spatial and temporal domains. Based on this observation, we can define dimension projection operator $\Psi$ as follows:

$$\Psi^t([[\cdot]]) = \{x | x \text{ is a dimension and } c(x) \text{ is true}\}$$

where, $t$ is a target dimension and $c$ is a condition. To apply the definition, let’s consider FOREST_FIRE relation. Figure 9 shows extents of ‘The Big Fire’ associated with temporal element $t_i$ and Figure 10 shows the relation in the parametric data model. Note that $t_i$ is a temporal element, not a time instance.

![Figure 9: Fire extents of ‘The Big Fire’](image)

<table>
<thead>
<tr>
<th>FOREST_FIRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firename</td>
</tr>
<tr>
<td>$s_1 \times t_1$ The Big Fire</td>
</tr>
<tr>
<td>$\cup s_2 \times t_2$</td>
</tr>
<tr>
<td>$\cup s_3 \times t_3$</td>
</tr>
<tr>
<td>$\cup s_4 \times t_4$</td>
</tr>
</tbody>
</table>

Figure 10: FOREST_FIRE relation in the parametric data model for Figure 9

Suppose that a user poses a query asking the largest extent of ‘The Big Fire.’ Then we can retrieve the largest extent of the fire by using the dimension projection as follows:

$$\Psi^S_{\maxarea}([[\text{Firename = ‘The Big Fire’}]]) = \{s_4\}$$
The execution of the dimension projection consists of three steps: 1) extract a parametric domain, 2) project specified domain, and 3) apply conditions.

By the first step of the dimension projection, a spatiotemporal element is extracted as follows:

$$[[\text{Firename} = \text{‘The Big Fire’}]] = s_1 \times t_1 \cup s_2 \times t_2 \cup s_3 \times t_3 \cup s_4 \times t_4$$

By the second step, only spatial elements are projected from the spatiotemporal element as follows:

$$\Psi^S([[\text{Firename} = \text{‘The Big Fire’}]] = \{s_1, s_2, s_3, s_4\}$$

By the last step, only qualified spatial elements are returned as follows:

$$\Psi^S_{\text{maxarea}}([[\text{Firename} = \text{‘The Big Fire’}]] = \{s_4\}$$

The ParaSQL query uses the sublanguage defined for navigating the internal parametric elements of tuples. The definition and the grammar of the language are not formally defined yet and should be studied. But the definition of the dimension projection that we have discussed is sufficient to solve the problem of this example. Let’s define the format of the sublanguage tentatively as follows:

```
PROJ {space, time} 
FROM <domain expression>
WHERE <condition>
```

The ParaSQL query first retrieves tuples such that the name of a fire is ‘The Big Fire’ and find the largest extent from the domain of the fire. In this case, the largest region will be returned. Since the return value is wrapped up with domain expression in RESTRICTED TO clause, the underlying system of parametric database aligns the dimension with $$[[s_4]] \times T$$, where $$T$$ is the universal set of time. By the restriction, the domain of tuple will be intersected with the spatiotemporal element of the tuple. Therefore, we can get the time and the largest extent of the fire.

2. When and where was the spread of fires larger than 500 km²?

**SQLST:**

```
SELECT F1.VTime, F2.Extent 
FROM FOREST_FIRE as F1 F2
WHERE F1.VTime = F2.VTime 
AND F1.Firename = F2.Firename
GROUP BY F1.VTime, F2.Extent, F1.Firename
HAVING AREA(F1.Extent) > 500
```

3. Determine the times and locations when “The Big Fire” started.

**SQLST:**

```
SELECT VTime, Extent 
FROM FOREST_FIRE 
WHERE Firename = ‘The Big Fire’ 
AND VTime = (SELECT MIN(VTime) 
FROM FOREST_FIRE 
WHERE Firename = ‘The Big Fire’)
```
3.3  E.S STQL

3.3.1  E.S STQL Data Model

E.S STQL stands for Erwig and Shneider’s Spatio-Temporal Query Language\textsuperscript{12}. E.S STQL models spatiotemporal data as abstract data types which can be employed as attribute types in a relation. The objective of E.S STQL data model is to be compatible with smoothly changing spatiotemporal objects. The temporal version of a value of type $\alpha$ that changes over time is modeled as a \textit{temporal function} of type $\tau(\alpha) = \text{time} \rightarrow \alpha$, where $\alpha$ is a spatial data type such as $\text{point}$ and $\text{region}$. Temporal functions are basis of an algebraic data model for spatiotemporal data types. Therefore, $\tau(\text{point})$ represents a point changing its location over time. Similarly, an element of type $\tau(\text{region})$ is a region that can move and/or grow/shrink. In addition, E.S STQL data model also has changing numbers and booleans when defining operations on temporal objects. For example, if we want to compute the time dependent distance of an airplane and a storm. This could be achieved by an operator:

$$\text{Distance} : \tau(\text{point}) \times \tau(\text{region}) \rightarrow \tau(\text{real})$$

The data model of E.S STQL has a special operator \textit{lift} to make non-temporal operation work on temporal objects and return a temporal object as a result. For non-temporal function $f : \alpha_1 \times \cdots \times \alpha_n \rightarrow \beta$, its corresponding lifted versions defined as follows:

$$\uparrow f : \tau(\alpha_1) \times \cdots \times \tau(\alpha_n) \rightarrow \tau(\beta)$$

with

$$\uparrow f(S_1, \cdots, S_n) := \{(t, f(S_1(t), \cdots, S_n(t))) | t \in \text{time}\}$$

For example, consider the spatial predicate $\text{inside} : \text{point} \times \text{region} \rightarrow \text{bool}$. The lifted version of this predicate has the type

$$\uparrow \text{inside} : \text{Point} \times \text{Region} \rightarrow \text{Bool}$$

with the meaning that it yields \text{true} for each time at which the point is inside the region, \text{undefined} whenever the point or the region is undefined, and \text{false} in all other cases\textsuperscript{13} \cite{45}.

3.3.2  Syntax of E.S STQL

E.S STQL extends the widespread database query language SQL. There is no formal syntax definition on E.S STQL in literature. The following syntax is derived based on the examples in \cite{45} and the data model of E.S STQL. Since it adapts the standard SQL, the main skeleton of the syntax will be SELECT-FROM-WHERE clause and SELECT clause may have attributes forming temporal functions of type.

SELECT <attribute list> | <temporal function of type>
FROM <relation list>
WHERE <where condition>
[GROUP BY <group by clause>]
[HAVING <having clause>]

\textsuperscript{12}Martin Erwig and Markus Shneider developed a spatiotemporal query language, and they named it STQL. During the literature survey, I found that there existed another spatiotemporal query language with the same name. In this paper, E.S STQL is the spatiotemporal query language developed by Erwig and Shneider.

\textsuperscript{13}According to \cite{45}, the authors denote non-temporal types, entities, functions, and predicates by lower case letters while their temporal counter parts start with capital letters to make notations more comprehensible.
3.3.3 Examples of Queries

For comparisons, let’s define FLIGHT and WEATHER relations as follows:

\[
\begin{align*}
\text{FLIGHTS(ID: string, Route: Point)} \\
\text{WEATHER(Kind: string, Extent: Region)}
\end{align*}
\]

The attribute ID identifies a flight, and Route records the route of a flight over time. The attribute Kind classifies different weather events like hurricanes, high pressure areas, or snowfall, and Extent yields the evolving extent of each weather event.

1. Where was United Airlines flight 207 at time 8:00?

E.S STQL:

\[
\begin{align*}
\text{SELECT Route(8:00)} \\
\text{FROM FLIGHTS} \\
\text{WHERE ID = ‘UA207’}
\end{align*}
\]

In E.S STQL, it retrieves Route attribute with a time argument. Based on the data model used in E.S STQL, the attribute value is the function of time as defined below:

\[
\text{Route : time } \rightarrow \text{ Point}
\]

Therefore Route(8:00) returns the point of airline ‘UA 207’ at 8:00.

2. When was a plane over the Eiffel Tower?

E.S STQL:

\[
\begin{align*}
\text{SELECT dom(Intersection(Route, ^EiffelTower))} \\
\text{FROM FLIGHTS}
\end{align*}
\]

ParaSQL:

\[
\begin{align*}
\text{SELECT F.ID} \\
\text{RESTRICTED TO [EiffelTower]} \\
\text{FROM FLIGHTS F}
\end{align*}
\]

In E.S STQL, the lifting operator is denoted by ^\. The Intersection operator is lifted and computes the time-dependent intersection of two moving points. The result is a moving point comprising all those (time, point)-pairs where the two original moving points met.

In the above E.S STQL and ParaSQL, they use EiffelTower to describe a point containing the coordinates of the Eiffel Tower. Since there is no relation on Eiffel Tower, we have to assume that a user knows the region (here, point) of the tower. The ParaSQL query retrieves the parametric elements (the region of Eiffel Tower). If the result is non-empty after intersecting two domains, the flight was over the tower at a specific time. Here, we have to note the domain alignment. Two query languages have different approach to align domains. In E.S STQL, users have to handle the domain alignment by using the lift operator, but in ParaSQL, the work is left to a underlying system. Therefore, in E.S STQL users are required to determine which attributes are time independent before asking queries.
3. Determine the time when flight UA207 flew into a hurricane

E.S STQL:

```sql
SELECT MIN(dom(Intersection(Route, Extent)))
FROM FLIGHTS, WEATHER
WHERE ID = 'UA207' AND Kind = 'hurricane'
```

3.4 K.R.P STQL

3.4.1 K.R.P STQL Data Model

The data model of K.R.P STQL\(^{14}\) supports a bitemporal concept for a spatial object such as point, line, and polygon objects. In this model, an object is represented using a hierarchical three-dimensional architecture that consists of two-dimensional space domain and linear valid time domain on the basis of another time domain, entitled as linear transaction time domain. Figure 11 shows the logical spatiotemporal database and describes a table that consists of primitive, spatial, valid time, and transaction time attributes. The spatial attribute has one or more values; the point type has a pair of spatial coordinates, i.e., \((x, y)\). The line type has two pairs of spatial coordinates that stand for the first and final points, respectively. Also, the polygon type has a sequence of points. K.R.P STQL data model uses the spatiotemporal first normal form (ST-1NF) sustaining a spatial attribute to have one or more values. [46, 47].

<table>
<thead>
<tr>
<th>Primary Key</th>
<th>Primitive attribute (char, int, bool)</th>
<th>Spatial attribute (point, line, polygon)</th>
<th>Temporal attribute (valid, transaction time)</th>
</tr>
</thead>
</table>

Figure 11: Table structure of STDB [47]

Let’s consider the following relations—BUILDING and CABLE. This BUILDING relation has Name, Owner, Price, Location, and Shape attributes. Especially, Location and Shape attributes are point and polygon types, respectively. CABLE relation has Name, Manhole, and Section attributes. Section attribute is line type. Since K.R.P STQL has been built upon a bitemporal model, there are additional columns such as VF, VT, TS, and TE. The synonym VF stands for valid from time. Also the VT, TS, and TE mean valid to, transaction start, and transaction end, respectively [47].

```sql
BUILDING (Name, Owner, Price, Location, Shape, VF, VT, TS, TE)
CABLE (Name, Manhole, Section, VF, VT, TS, TE)
```

Figure 12 and Figure 13 shows snapshot tables for BUILDING and CABLE relations\(^{15}\).

3.4.2 Syntax of K.R.P STQL

K.R.P STQL is designed on the basis of SQL3 and TSQL2 and the syntax is defined as follows:

```sql
SELECT attribute_name [, attribute_name]*
FROM table_name [AS alias_name][, table_name [AS alias_name]]*
```

\(^{14}\)K.R.P STQL is the spatiotemporal query language developed by the authors in [46, 47]. They named it STQL, but there already exists another STQL introduced in the previous section, in this paper we call it K.R.P STQL.

\(^{15}\)In order to simplify the date, we listed the date shown in the table in [47], and gave unique numbers to every date.
<table>
<thead>
<tr>
<th>Name</th>
<th>Owner</th>
<th>Price</th>
<th>Address</th>
<th>Location</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Book</td>
<td>Jane</td>
<td>100,000</td>
<td>Gaeshin 25</td>
<td>(100,100)</td>
<td>((90,90),(90,110),(110,110),(110,90))</td>
</tr>
<tr>
<td>KFC</td>
<td>Tom</td>
<td>35,000</td>
<td>Sajik 77</td>
<td>(50,50)</td>
<td>((40,40),(40,60),(60,60),(60,40))</td>
</tr>
<tr>
<td>KFC</td>
<td>Tom</td>
<td>35,000</td>
<td>Sajik 77</td>
<td>(50,50)</td>
<td>((40,40),(40,60),(60,60),(60,40))</td>
</tr>
<tr>
<td>MacDonald</td>
<td>Nick</td>
<td>70,000</td>
<td>Sajik 77</td>
<td>(50,50)</td>
<td>((40,40),(40,60),(60,60),(60,40))</td>
</tr>
<tr>
<td>Lotteria</td>
<td>Jane</td>
<td>90,000</td>
<td>Gakyung 45</td>
<td>(200,200)</td>
<td>((190,190),(190,210),(210,210),(210,190))</td>
</tr>
<tr>
<td>Lotteria</td>
<td>Jane</td>
<td>90,000</td>
<td>Gakyung 115</td>
<td>(300,300)</td>
<td>((290,290),(290,310),(310,310),(310,290))</td>
</tr>
<tr>
<td>Lotteria</td>
<td>Jane</td>
<td>90,000</td>
<td>Gakyung 115</td>
<td>(300,300)</td>
<td>((290,290),(290,310),(310,310),(310,290))</td>
</tr>
<tr>
<td>Lotteria</td>
<td>Jane</td>
<td>90,000</td>
<td>Gakyung 115</td>
<td>(300,300)</td>
<td>((190,190),(190,410),(410,410),(410,190))</td>
</tr>
</tbody>
</table>

Figure 12: BUILDING table snapshot [47]
<table>
<thead>
<tr>
<th>Name</th>
<th>Manhole</th>
<th>Section</th>
<th>VF</th>
<th>VT</th>
<th>TS</th>
<th>TE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heungdok-1</td>
<td>(120, 80)</td>
<td>((10,10),(20,50),</td>
<td>0</td>
<td>NOW</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(30,40),(150,190))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heungdok-1</td>
<td>(120, 80)</td>
<td>((10,10),(20,50),</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(30,40),(150,190))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heungdok-1</td>
<td>(120, 80)</td>
<td>((10,10),(20,50),</td>
<td>5</td>
<td>NOW</td>
<td>5</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(30,40),(110,105),</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(140,135), (150,190)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heungdok-2</td>
<td>(40, 275)</td>
<td>((10,310),(20,250),</td>
<td>6</td>
<td>NOW</td>
<td>6</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(30,280),(70,390))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 13: CABLE table snapshot [47]

The SELECT statement is a major one in DML (Data Manipulation Language) and it retrieves the spatiotemporal information for objects from the past to current using specified temporal and spatial predicates. The target list of the select statement is filled with attributes to be displayed or stored into another table. An alias name for table can be used in the FROM clause. The VALID clause is used to specify the time to be displayed for retrieved tuples. The WHERE clause includes the relational and spatial predicates. It works on primitive and spatial attributes, respectively. The spatiotemporal predicates are described in WHERE and WHEN clauses. In queries, the spatial operation AREA and temporal operation VALID can be used as a function type [46, 47].

3.4.3 Examples of Queries

1. Retrieve the building name from BUILDING table that had been built before 1 March 1985\(^{16}\) and is over 50m\(^2\).

   **K.R.P STQL:**
   ```sql
   SELECT Name
   FROM BUILDING AS B
   WHEN BEGIN(VALID(B)) PRECEDE TIMESTAMP 1985-03-01
   WHERE AREA(B.Shape) > 50
   ```

2. Retrieve the manhole that was built within 1 year from now and at 1 km distance from the line across the building. The building should have been built before 1 March 1985 and is over 50m\(^2\).

   **K.R.P STQL:**
   ```sql
   SELECT C.Manhole
   ```

\(^{16}\)1985/03/01 has been denoted as 2 in the relational tables.
FROM BUILDING AS B, CABLE AS C
WHEN BEGIN(VALID(B)) PRECEDE TIMESTAMP 1985-03-01
AND BEGIN(VALID(C)) PRECEDE BIND(NOW-1 Year)
WHERE AREA(B.Shape) > 50 AND B.Shape CROSS C.Section
AND DISTANCE(B.Location, C. Manhole) <= 1 km

3.5 STSQL

3.5.1 STSQL Data Model

Authors in [48] extended SQL to spatiotemporal SQL (STSQL) and it is based on a temporally
extended SQL, termed ATSQL [51]. STSQL supports the two temporal aspects, valid time and
transaction time. STSQL has been designed at TimeCenter\textsuperscript{17} and the purpose of the query language
is to support spatiotemporal query over spatiotemporal databases. In order to naturally general-
ize the snapshot relational model to a dimensional relational model, they adopt the view that a
dimensional table simply is a collection of snapshot tables, with each snapshot table having an
associated multi-dimensional point and containing all the snapshot tuples that have an associated
multi-dimensional region that contains the point.

STSQL introduces new datatypes that capture time and space values. For time values STSQL
uses anchored time periods. Spatial values are unions of regions. Regions are either defined over
1-, 2-, or 3-dimensional spatial domains. The corresponding datatypes are PERIOD, 1D_REGION,
2D_REGION, and 3D_REGION, respectively.

The below defines three relations-STANDS, ESTATES, and PLANS.

\begin{align*}
\text{STANDS} & (\text{STID}, \text{Index}, \text{Specie}, \text{Planted}, \text{Summary}, \text{STVT}, \text{STTT}, \text{StArea}) \\
\text{ESTATES} & (\text{ESID}, \text{Owner}, \text{ESArea}, \text{ESVT}, \text{ESTT}) \\
\text{PLANS} & (\text{PLID}, \text{STID}, \text{Volume}, \text{Ripe}, \text{PLVT}, \text{Harvest1}, \text{Harvest2})
\end{align*}

In these relations, VT and TT represent a valid time and a transaction time of a tuple, respec-
tively. In [48], the authors alter base tables for STANDS, ESTATES, and PLANS to add columns
representing temporal and spatial information.\textsuperscript{18} But it is basically same work to create tables
based on the relation schemas.

Figure 14 shows the tables for the relations, and the descriptions are as follows [48]:

1. The STANDS table models the (surveyed and analyzed) status of stands. For each stand,
   the specie of the stand’s dominant tree population, the soil fertility of the stand, the stand’s
   location, and a period of validity are recorded.

2. The ESTATES table records for each estate its owner, the validity period of the ownership,
   and the area that it covers.

3. The PLANS table records how stands are cultivated. For each stand, the volume to be
   harvested and the ripe year are recorded. Each plan has two harvest periods, calculated
   according to different scheduling methods that emphasize some growth conditions differently,
   e.g., according to soil fertility, climate, etc.

\textsuperscript{17}http://www.cs.auc.dk/general/DBS/tdb/TimeCenter
\textsuperscript{18}The base relation does not have VT and TT columns. They are ordinary relational tables.
<table>
<thead>
<tr>
<th>STID</th>
<th>Index</th>
<th>Specie</th>
<th>Planted</th>
<th>Survey</th>
<th>STVT</th>
<th>STTT</th>
<th>StArea</th>
</tr>
</thead>
<tbody>
<tr>
<td>st100</td>
<td>high</td>
<td>pine</td>
<td>1935</td>
<td>1984-1986</td>
<td>1989-NOW</td>
<td>1996-NOW</td>
<td>regst100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ESID</th>
<th>Owner</th>
<th>ESArea</th>
<th>ESVT</th>
<th>ESTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>es34</td>
<td>Paul</td>
<td>reges34</td>
<td>1995-NOW</td>
<td>1994-NOW</td>
</tr>
<tr>
<td>es63</td>
<td>Mary</td>
<td>reges63</td>
<td>1996-NOW</td>
<td>1996-NOW</td>
</tr>
<tr>
<td>es80</td>
<td>Peter</td>
<td>reges80</td>
<td>1996-NOW</td>
<td>1995-1996</td>
</tr>
<tr>
<td>es401</td>
<td>Mary</td>
<td>reges401</td>
<td>1996-NOW</td>
<td>1995-1996</td>
</tr>
<tr>
<td>es80</td>
<td>Peter</td>
<td>reges80</td>
<td>1996-1999</td>
<td>1997-NOW</td>
</tr>
<tr>
<td>es401</td>
<td>Mary</td>
<td>reges401</td>
<td>1996-1999</td>
<td>1997-NOW</td>
</tr>
<tr>
<td>es100</td>
<td>Tom</td>
<td>reges100</td>
<td>2000-NOW</td>
<td>1997-NOW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLID</th>
<th>STID</th>
<th>Volume</th>
<th>Ripe</th>
<th>PLVT</th>
<th>Harvest1</th>
<th>Harvest2</th>
</tr>
</thead>
</table>

Figure 14: The spatiotemporal example database [48]

### 3.5.2 Syntax of STSQL

STSQL follows the syntax of SQL. In addition, it has a flag before SQL statements are described. Flag is the central construct and used to indicate the desired evaluation mode(s) such as whether the statements have to be evaluated sequentially and/or non-sequentially. The following EBNF defines the syntax of flag. The `<cursor specification>` is the standard’s production for the SELECT statement.

```plaintext
<cursor specification> ::= flags <query expressions> [ <order by clause>]
flags ::= [flag { "AND" flag } | range_spec { "AND" range_spec}]
flag ::= modifier dimensions [ domain,constant | range_spec]
range_spec ::= "SET" <identifier> dim,datatype range_expression
modifier ::= "SEQUENCED" | "NONSEQUENCED"
dimensions ::= "(" column_reference { column_reference | "")" | "AS" <identifier>]
```
dim,datatype ::= PERIOD | 1D_REGION | 2D_REGION | 3D_REGION

SEQUENCED string indicates how to handle the dimension attributes in the queries and also restricts the qualifying tuples based on the values of their dimension attributes. NONSEQUENCED indicates dimension attributes that should be treated as regular attributes in a query [48].

3.5.3 Examples of Queries

1. For each stand that is ripe in 2000, determine its harvest periods.

STSQL:

```sql
> SEQUENCED (STVT, PLVT) AS VT AND NONSEQUENCED (Harvest1, Harvest2)
SELECT Harvest1, Harvest2
FROM STANDS ST, PLANS PL
WHERE PL.STID = ST.STID
AND ST.Ripe = 2000;
```

2. Determine all stands that did not change status for more than 5 years, together with the corresponding estate(s).

STSQL:

```sql
> SEQUENCED (STArea, ESArea) AS STESArea AND
SEQUENCED (STVT, ESVT) AS STESVT
SELECT STID, ESID
FROM STANDS, ESTATES
WHERE DURATION(STVT, YEAR) > 5;
```

3. For all stands, determine when the two harvest periods are scheduled contemporary.

STSQL:

```sql
> SEQUENNCED (PL1.Harvest1, PL2, Harvest2) AS AgreedHarvest
AND NONSEQUENCED (PL1.Harvest2, PL2.Harvest1)
SELECT PL1.STID
FROM PLANS PL1, PLANS PL2
WHERE PL1.STID = PL2.STID;
```

4 Summary

There has been considerable research on general temporal databases and on spatial databases [40]. However, temporal databases and spatial databases have long been separate and important areas of database research, and researchers in both areas have felt that there are important connections in the problems addressed by each area, and the techniques and tools utilized for their solution [49].

In this chapter, we have discussed four different spatiotemporal data model and extended SQL style spatiotemporal query languages-SQL$^{ST}$, E.S. STQL, K.R.P STQL, and STSQL as well as compared them with ParaSQL. All these query languages have their specific features handling problems that the authors in the literature had encountered.
SQL$^{ST}$ is the minimal extension from the standard SQL and it can provide users much similar standard SQL style spatiotemporal query language. It, however, cannot grasp the specific features of spatiotemporal data with natural and convenient ways because of the restriction of the standard SQL for spatiotemporal query.

E.S STQL defined attribute values as temporal functions of types. Therefore, the query language is very suitable for applications to query moving objects over time. One disadvantage of E.S STQL is that users have to handle the dimension alignment between two different domains by using lift operation.

K.R.P STQL has been built based on TSQL2 data model incorporated with spatial attributes. As mentioned in [50], TSQL2 proposal was met with little success. Since the model in K.R.P STQL stores an object in different tuples, it requires to do a self-join to process queries on the object.

STSQL was proposed in reference [51] and developed at TimeCenter. Because space and time are captured by separate attributes, STSQL is intended for applications that do not involve storing the movement of continuously moving objects [30]. Since STSQL data model is also based on TSQL2 data model, it has the same drawbacks that K.R.P STQL has.
References


