A Model Comparison for Spatiotemporal Data in Ubiquitous Environments: A Case Study

Seo-Young Noh* and Shashi K. Gadia**

Abstract—In ubiquitous environments, many applications need to process data with time and space dimensions. Because of this, there is growing attention not only on gathering spatiotemporal data in ubiquitous environments, but also on processing such data in databases. In order to obtain the full benefits from spatiotemporal data, we need a data model that naturally expresses the properties of spatiotemporal data. In this paper, we introduce three spatiotemporal data models extended from temporal data models. The main goal of this paper is to determine which data model is less complex in the spatiotemporal context. To this end, we compare their query languages in the complexity aspect because the complexity of a query language is tightly coupled with its underlying data model. Throughout our investigations, we show that it is important to intertwine space and time dimensions and keep one-to-one correspondence between an object in the real world and a tuple in a database in order to naturally express queries in ubiquitous applications.

Keywords—Parametric Data Model, Model Comparison, ParaSQL, Spatiotemporal Data

1. INTRODUCTION

Data has to be gathered and processed in ubiquitous environments naturally and effectively. Data arising in such environments has time and space dimensions that heavily require database supports in order to create valuable information from gathered data. Because of this, spatiotemporal databases have potential benefits in ubiquitous applications like environment monitoring, health care, smart-homes, and real-time sensor data analysis [1, 2]. Environment monitoring systems, for example, are scattered in hazardous areas and send data in a spatiotemporal context to a centralized repository and data in the repository tend to be processed in real time.

Research on spatiotemporal databases has been independently done in the temporal database community and spatial database community, respectively [3]. Therefore there are two approaches to spatiotemporal data models: one from a temporal data model and the other from a spatial data model [4, 5]. In this paper, we focus on the former approach.

In general, temporal data models can be categorized into three classes depending on the types of data handling—*point-based* [6], *interval-based* [7, 8], and *temporal element-based* [9, 10] data models. In a point-based data model, a time instant is associated with a tuple and all attributes in the tuple share the same time instant. In an interval-based data model, a time inter-

Manuscript received April 27, 2011; accepted September 23, 2011.
Corresponding Author: Seo-Young Noh
* Supercomputing Center, Korea Institute of Science and Technology Information, Daejeon, Korea (rsyoung@kisti.re.kr)
** Department of Computer Science, Iowa State University, Ames, IA 50014, USA (gadia@cs.iastate.edu)

val is used to validate a tuple. Similarly to the point-based data model, all attributes in a tuple share the same time interval. In a temporal element-based data model, a temporal element is used to identify the domain of a temporal tuple and is defined as a finite union of intervals.

We can also classify these three models into two groups depending on the way of timestampings: *tuple-level timestamping* and *attribute-level timestamping*. The point-based and the interval-based data models use tuple-level timestamping while the temporal element-based data model uses attribute-level timestamping. All attributes in tuple-level timestamping share their temporal domains or spatiotemporal domains. This feature makes the two data models popular in temporal databases because conventional databases can be utilized for their implementation.[1]

However, conventional database systems are not suitable for spatiotemporal data generated in ubiquitous environments because the data model in conventional database systems mainly focus on capturing the current perception of reality, not the history of data.

In this paper, we introduce three spatiotemporal query languages such as SQL^{ST}, STSQL, and ParaSQL. SQL^{ST} is a spatiotemporal query language introduced by Chen and Zaniolo [13] and it is based on the point-based temporal data model with tuple-level timestamping. This type of query language is minimally extended from classical SQL. Guting et al. [14] introduced a spatiotemporal query language and proposed spatiotemporal data types, which are time-dependent geometries. Such a query language is SQL-like and we named it STSQL (SpatioTemporal SQL) for our query language comparisons. We use their temporal data types for the interval-based data model and assume that STSQL is implemented on top of a conventional database system.[2] ParaSQL [15, 16] is a spatiotemporal query language, which uses temporal elements (or spatiotemporal elements) for domains.

In this paper, we use Guting's use case in order to compare the complexity of the three query languages. Although the use case cannot cover all aspects of spatiotemporal data models, they are sufficient for understanding the fundamental differences of the three data models because a query language cannot be independent of its underlying data model. Throughout our comparisons, we show that ParaSQL expresses the use case much simpler than the others and that it also requires less disk accesses.

The rest of this paper is organized as follows: in Section 2, we introduce time representations and conceptual temporal relations in the three data models. In Section 3, we discuss the query language syntaxes of the data models. In Section 4, we compare the three spatiotemporal query languages based on Guting's use case. Finally, we conclude our paper in Section 5.

2. TIME REPRESENTATIONS AND TEMPORAL RELATIONS

2.1 Time Representations

In our comparisons, discrete linearly ordered models of time are considered. Let T be a set of

1 Two data models are different, but they have the same viewpoint in implementations. Chomicki [11] viewed temporal databases in two classes: abstract temporal databases and concrete temporal databases. Terenziani and Snodgrass [12] introduced the concepts—*telic* and *atelic*--in capturing the distinction between point-based and interval-based approaches. *Telic* means accomplishments, while *atelic* means states. Toman [6] says that the two classes share the same underlying structure of time instants and provided a mapping mechanism from the class of concrete databases to abstract temporal databases. Therefore, for every interval-based query, there is an equivalent point-based query.

2 It should be emphasized that the attribute types introduced by Guting et al. are orthogonal to underlying data models and STSQL is only one of the possible query languages.

time instants and a discrete linear order on T.

The interval I is a subset of T such that whenever $t_1 \in I$, $t_2 \in I$ and $t_1 < t_3 < t_2$, then $t_3 \in I$. A temporal element E is a finite union of intervals. In other words, E is a temporal element if the intervals $I_1 \cdots I_n$ exist in a way such that $E = I_1 \cup \cdots \cup I_n$. For example, $[2, 5] \cup [7, 10]$ and $[2, 5] \cup [4, 10] \cup [30, 40]$ are temporal elements. Note that the latter one can also be represented as $[2, 10] \cup [30, 40]$. In general, a temporal element can be expressed uniquely as a finite union of mutually disjoint intervals.

It is worth noting that a set of time instants and a temporal element are closed under set theoretic operations like union, intersection, and difference.

2.2 Temporal Relations

In this subsection, we will discuss how to represent temporal relations in the three temporal data models. For the sake of simplicity, we use the terms—SQLST, STSQL, and ParaSQL for point-based, interval-based, and temporal element-based data models, respectively. In our discussion, we use the information from Hurricane Charley [17], which was measured in 2004, as shown in Fig. 1.

SQLST is point-based. Therefore, data in a temporal relation is interpreted as a sequence of states indexed by points in time. Fig. 1-(a) shows a conceptual representation of Hurricane Charley in the point-based temporal data model. As shown in every time instant, the information about Hurricane Charley should be recorded in the table. We may achieve some storage efficiency by compressing internal data. However, we only consider conceptual representation ra-

Name	Stage	WindSpeed	T
Charley	Tropical Depression	$x < 50$	0
Charley	Tropical Depression	$x < 50$	1
Charley	Tropical Storm	$x < 50$	2
Charley	Tropical Storm	$50 \leq x < 100$	3
Charley	Tropical Storm	$50 \leq x < 100$	4
Charley	Hurricane	$100 \leq x$	5
Charley	Hurricane	$100 \leq x$	6
Charley	Hurricane	$100 \leq x$	7
Charley	Tropical Storm	$50 \leq x < 100$	8
Charley	Extratropical	$50 \leq x < 100$	9
Charley	Extratropical	$x < 50$	10

(a) SQLST

Name	Stage	WindSpeed	TS	TE
Charley	Tropical Depression	$x < 50$	0	1
Charley	Tropical Storm	$x < 50$	2	2
Charley	Tropical Storm	$50 \leq x < 100$	3	4
Charley	Hurricane	$100 \leq x$	5	7
Charley	Tropical Storm	$50 \leq x < 100$	8	8
Charley	Extratropical	$50 \leq x < 100$	9	9
Charley	Extratropical	$x < 50$	10	10

(b) STSQL

*TS: Start Time
*TE: End Time

Name	Stage	WindSpeed
[0,10] Charley	[0,1] Tropical Depression	[0,2]∪[10,10] $x < 50$
	[2,4] ∪ [8,8] Tropical Storm	[3,4]∪[8,9] $50 \leq x < 100$
	[5,7] Hurricane	[5,7] $100 \leq x$
	[9,10] Extratropical	

(c) ParaSQL

Fig. 1. Temporal relations in the three data models

ther than actual implementation in this paper.

We assumed that STSQL is interval-based. In the data model, a tuple is associated with a time interval so that the tuple is valid during the time period. The interval represents the start time and the end time of the validity.

Fig. 1-(b) shows the conceptual representation of Hurricane Charley in the interval-based data model. An interval is represented by two columns, TS and TE for starting time and ending time, respectively. Note that whenever an attribute state is changed from one to another, it results in creating a tuple. For example, when Hurricane Charley's stage has been changed from Tropical Depression to Tropical Storm, <Stage, 'Tropical Depression'> →<Stage, 'Tropical Storm'>, the information should be recorded in the Hurricane relation.

ParaSQL is temporal element-based. Fig. 1-(c) shows Hurricane relation in ParaSQL. In this data model, an attribute is defined as a function of time. Because of this, this type of data model can encapsulate an object in the real world into a single tuple in a database. For example, the information about Hurricane Charley is gathered in a single tuple. Keeping one-to-one correspondence between an object in the real world and a tuple in database is the most fundamental difference compared to the other data models.[3] Note that if we add spatial domains to Hurricane Charley, the temporal relations shown in Fig. 1 will be spatiotemporal relations.

3. QUERY LANGUAGE SYNTAXES

3.1 SQL^{ST}

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.

SQL^{ST} consists of two components, SQL^{T} and SQL^{S}, which support temporal information and spatial information, respectively. Such components are based on Worboy's suggestion introduced in [18]. SQL^{T} [19] supports valid-time queries by minimal extensions of SQL, while SQL^{S} uses a polygon-oriented representation.[4] SQL^{ST} views reality as a sequence of snapshots of objects. Fig. 2 shows a simplified BNF of SQL^{ST}. Because this language minimally extends classical SQL, its BNF is almost identical to the classical SQL.

```
<query>  ::=  SELECT  <attribute  list>
              FROM  <relation  list>
              [WHERE  <boolean  expression>]
              [GROUP  BY  <attribute  list>
              [HAVING  <boolean  expression>]]
```

Fig. 2. The BNF of SQL^{ST}

3 Note that objects in the temporal element-based data model are not the same as those in object-oriented paradigm. There are no concepts of polymorphism and inheritance.

4 A similar approach, which triangulates spatial information, can be found in [20,21].

3.2 STSQL

STSQL uses the attribute types, which are time-dependent geometries. Note that STSQL is one of the possible query languages that can embed the data types. Fig. 3 shows a simplified BNF of STSQL.

<assignments> can be a query assignment, a function assignment, or a conversion assignment. A query assignment assigns the results of the <query> to a new object called the <name>, which can be used in further steps of a query. A function assignment defines a new operator derived from existing ones. A conversion assignment makes it possible that a relation with a single tuple with a single attribute can be converted into a typical atomic value.

<query> is close to classical SQL. STSQL allows multiple queries separated by semicolons. Therefore a STSQL query may consist of multiple <assignments> and a <query>.

```
<assignments>  ::={LET  <name>  =  <query>}  |
                  {LET  <name>  =  <function  expression>}  |
                  {LET  <name>  =  <conversion>}

<function  expression>  ::= FUN(<parameter  list>)  <expression>

<conversion>  ::=  {ELEMENT  (<query>)}  |
                  {SET  (<attribute  name>,  <value>)}

<query> ::=SELECT  {<attribute  list>  | <derived  attribute>}
           FROM  <relation  list>
           [WHERE  <boolean  expression>]

<derived  attribute>  ::= <new  attribute  name>  AS  <expression>

<multiple  query>  ::= {<assignments>;}+  [<query>]
```

Fig. 3. The BNF of STSQL

3.3 ParaSQL

Fig. 4 shows a simplified BNF of ParaSQL. ParaSQL consists of three expressions: *relational expression*, *domain expression*, and *boolean expression*. They evaluate relations, spatiotemporal elements, and boolean values, respectively. These three expressions are mutually recursive, which means one expression may include another expression.

A relational expression is assigned by a select statement. It should be emphasized that ParaSQL uses spatiotemporal elements in spatiotemporal context, which are forms of *spatial element* × *temporal element*. Here, a spatial element satisfies the closure property and so does a spatiotemporal element.

A relational expression returns a relation that is a set of spatiotemporal tuples. The SELECT statement is similar to a classical SQL statement, but it has a RESTRICTED TO clause that restricts the domain of tuples qualified by <boolean expression> in the WHERE clause.

A domain expression, denoted as [[·]], is used to restrict the domain of tuples filtered by a boolean expression. The domain expression, [[<attribute>]], collects the spatiotemporal domain of an attribute. It also allows a nested relational expression so that the domain expression, [[<re-

```
<relational expression> ::=
        SELECT <attribute list>
        [RESTRICTED TO <domain expression>]
        FROM <relation list>
        [WHERE <boolean expression>]

<domain expression> ::=[[<attribute>]] |
        [[<attribute> op <attribute>]] |
        [[<attribute> op <value>]] |
        [[<relational expression>]] |
        <spatiotemporal element>

<boolean expression> ::=<domain expression> set_op
        <domain expression> |
        <attribute> op <attribute> |
        <attribute> op <value>
```

Fig. 4. The BNF of ParaSQL

lational expression>]], is a domain of the entire relation returned by <relational expression>.

A boolean expression has the same functionality as classical SQL, in that it either qualifies or disqualifies a tuple. But it differs from classical SQL because it can be constructed by domain expressions with set operations. For example, [[Stage='Tropical Storm']] $\neq \phi$ can be abbreviated to "Stage = Tropical Storm" in the WHERE clause, which means that sometime a hurricane's stage was a Tropical Storm.

4. QUERY COMPARISONS

Guting et al. [14] introduced the application: *Forest Fire Control Management*. To illustrate their model-independent attribute types, they provided English queries and expressed them in the STSQL. Chen and Zaniolo proposed their spatiotemporal data model and query language. In the explanation of their model and query language, they used the English queries and expressed them in the SQL^{ST}. Within this framework, we express the same English queries in ParaSQL for our comparisons. In the comparisons, we focus on the user-friendliness of the query languages as well as disk accesses required to process user queries.

4.1 Use Case: Forest Fire Control Management

In many countries, fire is one of the main agents of forest damage. The main purposes of this type of management system are to learn about past fires and their evolution and to prevent fires in the future by studying weather and other factors like cover type, elevation, slope, distance to roads, and distance to human settlements [14].

> *The Forest Lake Fire was reported to the park dispatcher by the Mt. Sheridan lookout at 1850 on August 29, 1981. South District Ranger Mernin located the fire on the ground and confirmed that it was lightning-caused. The air patrol reported the development of the fire as follows: from 0 to 3, 20 acres; from 10 to 50, 312 acres; from 13 to 15, 54 acres; and from 16 to 25, 331 acres burned* [22].

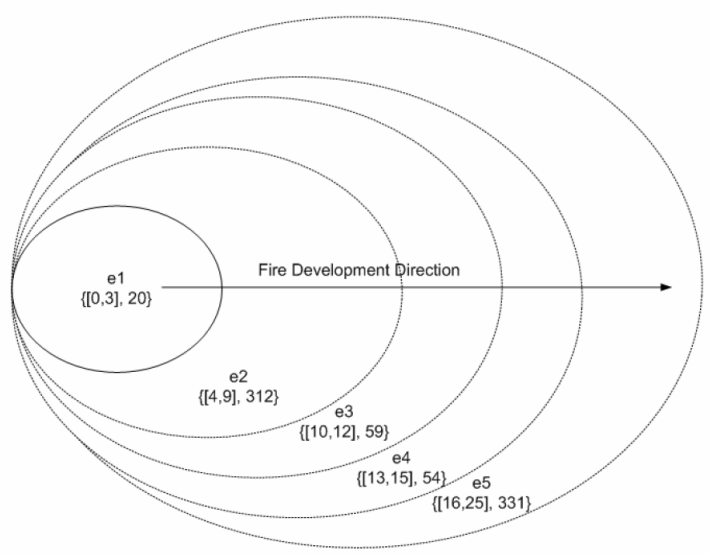

Fig. 5. Forest Lake Fire development

Fig. 5 shows the Forest Lake fire development. The fire's developments are represented as fire extents with a time period and the area burned per acre, where the fire extent is defined as *the area burned per time period or event* [23].

The following example illustrates forest fire control management. The Forest Lake Fire occurred in Wyoming in 1981.

4.2 Schemas and Spatiotemporal Relations

The schemas for this example are as follows:

SQLST:

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)

STSQL:

FOREST (Forestname:String, Territory:MRegion)
FOREST_FIRE (Firename:String, Extent:MRegion)
FIRE_FIGHTER (Fightername:String, Location:MPoint)

ParaSQL:

FOREST (Forestname)
FOREST_FIRE (Firename)
FIRE_FIGHTER (Fightername)

Firename	Extent	VTime
Forest Lake Fire	e_1	0
Forest Lake Fire	e_1	3
Forest Lake Fire	e_2	4
Forest Lake Fire	e_2	9
Forest Lake Fire	e_3	10
Forest Lake Fire	e_3	12
Forest Lake Fire	e_4	13
Forest Lake Fire	e_4	15
Forest Lake Fire	e_5	16
Forest Lake Fire	e_5	25

(a)SQLST

Firename	Extent
Forest Lake Fire	$[0,3] \times e_1$
Forest Lake Fire	$[4,9] \times e_2$
Forest Lake Fire	$[10,12] \times e_3$
Forest Lake Fire	$[13,15] \times e_4$
Forest Lake Fire	$[16,25] \times e_5$

(b) STSQL

Firename		Extent	
$[0,3] \times e_1$ $\cup[4,9] \times e_2$ $\cup[10,12] \times e_3$ $\cup[13,15] \times e_4$ $\cup[16,25] \times e_5$	Forest Lake Fire	$[0,3] \times e_1$	20
		$[4,9] \times e_2$	312
		$[10,12] \times e_3$	59
		$[13,15] \times e_4$	54
		$[16,25] \times e_5$	331

(c) ParaSQL

Fig. 6. FOREST_FIRE relations

The FOREST relation maintains spatiotemporal data for forests such as locations and developments of growing and shrinking over time. The FOREST_FIRE relation records the evolution of fires. The FIRE_FIGHTER relation describes the motions of fire fighters being on duty from their start at the first station up to their return [14].

The schemas in SQLST use the spatial data types of REGION and POINT for representing a region and a location, respectively. The schemas in STSQL use spatiotemporal data types MRegion and MPoint for representing a moving region and a moving location, respectively. In ParaSQL, an attribute is assigned with spatiotemporal domain with values. Therefore, there is a single attribute for each relation.

Fig. 6 shows the conceptual FOREST_FIRE relations of SQLST, STSQL, and ParaSQL for the Forest Lake Fire, where *e* represents spatial domain. Note that SQLST triangulates *e* into multiple triangles.

4.3 Queries

Query 1: Find the time and the place which "Forest Lake Fire" had its largest extent.

SQLST:

```
SELECT F1.VTime,F2.Extent,AREA(F1.Extent)
FROM FOREST_FIRE AS F1 F2
WHERE F1.Firename = "Forest Lake Fire"
    AND F2.Firename = "Forest Lake Fire"
    AND F1.VTime = F2.VTime
GROUP BY F1.VTime
HAVING AREA(F1.Extent) =
    (SELECT MAX(AREA(Extent))
    FROM FOREST_FIRE
    WHERE Firename="Forest Lake Fire")
```

STSQL:

```
LET ForestLakeFire = ELEMENT (
    SELECT Extent
    FROM FOREST_FIRE
    WHERE Firename = "Forest Lake Fire");
LET max_area =
    initial((atmax(area(ForestLakeFire)));
atinstant(ForestLakeFire, inst(max_area));
val(max_area)
```

ParaSQL:

```
[[SELECT *
RESTRICTED TO [[Max(Area(DomPrj(S,F.FireName)))]]
FROM ForestFire F
WHERE F.FireName = "Forest Lake Fire"]]
```

The SQL^{ST} query uses a user-defined spatial aggregate function named AREA and a built-in aggregate function MAX. Since the temporal data model of SQL^{ST} is point-based, to retrieve all information about the Forest Lake Fire, it must perform a 2-way self-join to group the tuples recorded in every time instants.

The STSQL query consists of four parts. First, it extracts all extents for the Forest Lake Fire and assigns the elements to the variable ForestLakeFire. Second, it finds the maximum area among the extents and assigns the result to the variable max_area. Note that the variable max_area contains the value and spatiotemporal domain. Third, it finds time instants when the extents of the Forest Lake Fire were maximum. Last, it extracts the value of the max_area to return the largest extent. We must note that the values pointed by the variables— ForestLakeFire and max_area—should be materialized in order to be used by another step of the query. Since spatiotemporal data is complex and amassed, materialization results in expensive disk accesses. In addition, since STSQL uses intervals, it fragments an object in the real world into multiple tuples. Self-joins are required to gather tuples for an object, which join the same relations. Although STSQL does not explicitly show self-joins, they are unavoidable and should be processed in the STSQL somehow.

The ParaSQL query uses the function DomPrj to project a specific domain from a spatiotemporal element. The prototype of DomPrj is as follows:

DomPrj(dimension, spatiotemporal element)

Dimension indicates which dimension should be projected from the *spatiotemporal element*. In the spatiotemporal context, it can either be space dimension or time dimension. Let Ψ be the domain projector. The function Ψ_S projects a spatial domain while Ψ_T projects a temporal domain. For example, Ψ_S ([[FireName]]) projects the spatiotemporal element [[FireName]] as follows, where FireName = Forest Lake Fire:

$$\Psi_S([\![\text{FireName}]\!]) = \begin{cases} [0,3] \times e_1 & \longrightarrow e_1 \\ [4,9] \times e_2 & \longrightarrow e_2 \\ [10,12] \times e_3 & \longrightarrow e_3 \\ [13,15] \times e_4 & \longrightarrow e_4 \\ [16,25] \times e_5 & \longrightarrow e_5 \end{cases}$$

It is worth discussing the return values of a function whose input is another function. Such circumstances frequently occur when data should be processed in advance by another function. Furthermore, such compose functions can lead to less complex query languages and express queries more naturally. For our discussion, suppose that there is a function *f(x)* shown in Fig. 7.

Let *g* be a function. A compose function $g \circ f$ can be designed into three different ways:

1.It returns values for given domains.
2.It returns domains such that $g \circ f$ is true.
3.It returns values as well as domains such that $g \circ f$ is true.

Suppose *g* is a Max function that finds the maximum values of *f(x)*. We can express Max(*f(x)*) as follows:

$$\begin{array}{lcll} \text{Max}_1(f(x)) & = & y_1 & : \text{Type 1} \\ \text{Max}_2(f(x)) & = & \{x_1, x_2\} & : \text{Type 2} \\ \text{Max}_3(f(x)) & = & \{(x_1, y_1), (x_2, y_1)\} & : \text{Type 3} \end{array}$$

In the ParaSQL query, we assume that Max is the function of type 3 and so is the Area. Type-3 functions can articulate ParaSQL in more precise terms by removing the notation of the domain expression ([[·]]).

The procedural step of the ParaSQL query is straightforward. It uses the function Area, which returns the area of a symbolic spatial point as well as a spatiotemporal domain. By feeding the pairs of an area and a domain to the function Max, the ParaSQL can find the maximum areas. Since the function Max is type-3, it contains the domain of the maximum areas. Therefore, the domain expression in the RESTRICTED TO clause is able to restrict qualified tuples to the spa-

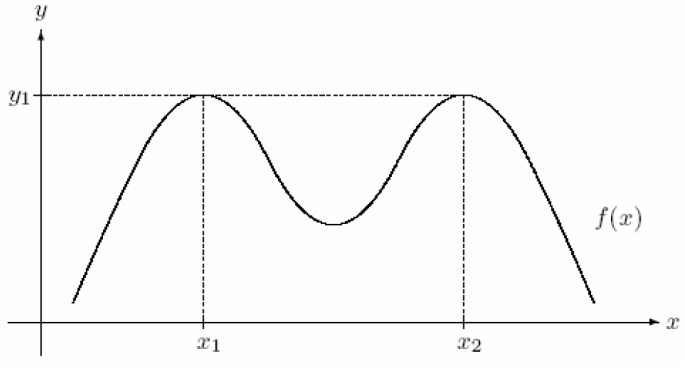

Fig. 7. Function *f(x)*

tiotemporal domain such that an extent is maximum. Since the result of the relational expression is a tuple, the ParaSQL takes [[·]] to extract the spatiotemporal domain for the tuple. Unlike SQL^{ST} and STSQL queries, the time complexity of this ParaSQL query is just a relation scan.

Query 2 When and where was the spread of fires larger than 500 km^2?

SQL^{ST}:

```
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
```

STSQL:

```
LET big_part =
    SELECT big_area AS extent
    WHEN [FUN (r:region) area(r) > 500]
    FROM FOREST_FIRE;

SELECT *
FROM big_part
WHERE not(isempty(deftime(big_area)))
```

ParaSQL:

```
[[SELECT *
    RESTRICTED TO [[
       Area(DomPrj(S,[[F.FireName]]))>500]]
    FROM ForestFire F]]
```

The FOREST_FIRE relation of SQL^{ST} shown in Fig. 6-(a) is a conceptual relation. When the relation is stored in a relational database, the model of SQL^{ST} triangulates spatial objects. Therefore, the tuples of FOREST_FIRE relation are actually stored as shown in Fig. 8.

In Fig. 8, (x_{ij},y_{ij}) represents a point of a triangle of extent e_i, where j indexes one of three points in a triangle. Due to the triangulations, SQL^{ST} needs self-joins to group extents based on their valid time.[5]

The STSQL query consists of two parts. The first part extracts extents such that the area of an extent is greater than 500 km^2 and assigns the set (or a relation) to variable big_part. The relation big_part has a single attribute whose name is big_area. The second part checks if the temporal domain of big_area is empty. Like Query 1, it also requires materialization.

In the ParaSQL query, it is similar to ParaSQL's Query 1. The domain expression, [[Area(DomPrj(S, F.FireName)) > 500]], restricts the domain of a tuple to a spatiotemporal

5 SQL^{ST} can avoid the self-join for this query if it stores un-partitioned spatial data in the relation instead of triangulating the data.

Name	Extent	x_1	y_1	x_2	y_2	x_3	y_3	Valid Time
Forest Lake Fire	e_1	x_{11}	y_{11}	x_{21}	y_{21}	x_{31}	y_{31}	0
Forest Lake Fire	e_1	x_{12}	y_{12}	x_{22}	y_{22}	x_{32}	y_{32}	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Forest Lake Fire	e_1	x_{1n}	y_{1n}	x_{2n}	y_{2n}	x_{3n}	y_{3n}	3
Forest Lake Fire	e_2	x'_{11}	y'_{11}	x'_{21}	y'_{21}	x'_{31}	y'_{31}	4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Fig. 8. Triangulation of spatial data in SQLST

element such that the size of the extent is greater than 500 km^2. If there are no such extents in a tuple (note that it is a single tuple), the domain will be empty and the tuple will be excluded. If there are such extents, the relational expression retrieves the extents. Since we need the spatiotemporal domain of the extents, the ParaSQL query takes [[·]] to extract the domain of the extents.

Query 3 Determine the time and location when the "Forest Lake Fire" started

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

SELECT VTime, Extent
FROM FOREST_FIRE
WHERE Firename = "Forest Lake Fire"
    AND VTime =
    (SELECT MIN(VTime)
    FROM FOREST_FIRE
    WHERE Firename = "Forest Lake Fire")
```

STSQL:

```
LET ForestLakeFire = ELEMENT (
    SELECT Extent
    FROM FOREST_FIRE
    WHERE Firename = "Forest Lake Fire");
LET start_extent = initial(min(
    deftime(ForestLakeFire)));
val(start_extent)
```

ParaSQL:

```
[[SELECT Firename
    RESTRICTED TO [[STARTTIME(Firename)]]
    FROM FOREST_FIRE
    WHERE Firename = "Forest Lake Fire"]]
```

The SQL^{ST} query first finds the minimum valid time of the fire in the WHERE clause. Since the English query requests the time and the location, SQL^{ST} should be nested.[6]

The STSQL query consists of three subqueries. First, it extracts all extents of the fire. Second it finds the first extent of the fire. Last, it returns the time and the location of the extent.

The ParaSQL query simply expresses the English query by restricting the tuple of the fire to the starting time. The domain of the fire is restricted to a spatiotemporal domain such that the time domain is the starting time instant and the spatial domain is the location at the time instant. This query requires only a single relation scan while the SQL^{ST} and STSQL query languages require additional relation scans.

Query 4 How long was fire fighter Th. Miller enclosed by the fire that was called the "Forest Lake Fire," and how much distance did he cover there?

SQL^{ST}:

```
SELECT DURATION(FIRE_FIGHTER.VTime),
    MOVING_DISTANCE(FIRE_FIGHTER.Location,  FIRE_FIGHTER.VTIME)
FROM FOREST_FIRE, FIRE_FIGHTER
WHERE FOREST_FIRE.VTime = FIRE_FIGHTER.VTime
    AND Firename = "Forest Lake Fire"
    AND Fightername = "Th. Miller"
GROUP BY FOREST_FIRE.VTime
HAVING INSIDE(Location, extent)
```

STSQL:

```
LET ForestLakeFire = ELEMENT (
    SELECT Extent
    FROM FOREST_FIRE
    WHERE Firename = "Forest Lake Fire");
LET start_extent = initial(min(
    deftime(ForestLakeFire)));
val(start_extent)

LET ForestLakeFire = ELEMENT (
    SELECT Extent
    FROM FOREST_FIRE
    WHERE Firename = "Forest Lake Fire");
```

6 The original query in [14] assumes that a fire can start at different time with different initial regions. However, the SQLST query introduced in [13] implicitly assumes that a fire started at a single time instant. For the sake of our discussion, we modified the STSQL query to make it follow the same assumption of SQL^{ST}.

```
SELECT time As duration(deftime(
                    intersection(location,ForestLakeFire))),
            distance As length(trajectory(
                    intersection(location, ForestLakeFire)))
    FROM FIRE_FIGHTER
    WHERE Fightername = “Th. Miller”
```

ParaSQL:

```
[[SELECT Firename
    RESTRICTED TO [[STARTTIME(Firename)]]
    FROM FOREST_FIRE
    WHERE Firename = “Forest Lake Fire”]]

SELECT DURATION([[Fightername]]),
        DISTANCE([[Fightername]])
RESTRICTED TO [[
    SELECT Firename
    FROM FOREST_FIRE
    WHERE Firename = “Forest Lake Fire”]]
FROM FIRE_FIGHTER
WHERE Fightername = “Th. Miller”
```

The SQLST query joins the two relations, FOREST_FIRE and FIRE_FIGHTER, based on valid times. It groups tuples by valid times and checks if Th. Miller's location is inside of the extents. For only those qualified tuples, it calculates the moving distance and the time length.

The STSQL query consists of two subqueries. The first subquery extracts all extents for Forest Lake Fire and assigns the elements to the variable ForestLakeFire. The second subquery retrieves a fire fighter tuple whose name is Th. Miller and calculates the length of distance and the duration such that he covered and he was enclosed by the fire, respectively.

The ParaSQL query retrieves a fire fighter tuple whose name is Th. Miller and restricts the domain of the tuple to the domain such that the Forest Lake Fire occurred, resulting in the intersection of the two spatiotemporal domains. It requires only relation scans unlike the others.

The procedural steps of STSQL and ParaSQL for this English query are very similar because two queries iterate the two different relations independently, avoiding join operations. However, if we closely investigate the underlying steps of STSQL, we will find that a join is involved. The variable ForestLakeFire is a relation name and it is scanned when intersecting with Location attribute of FIRE_FIGHTER relation. Since we assumed STSQL is using interval-based data model, for every location, all tuples of ForestLakeFire should be intersected. It is the same as a cartesian product. In the STSQL query, we can see there are two cartesian products to find the duration and the distance. In contrast, ParaSQL needs only 2 relation scans for the inner and the outer queries which are independent. Formally speaking, STSQL requires $O(n+2(n \cdot m))=O(n \cdot m)$ accesses while ParaSQL does $O(n+m)$, where n, and m are the number of tuples of Forest Lake Fire and that of the fire fighter Th. Miller.[7]

7 In ParaSQL, there are two tuples for the fire and the fire fighter. For brevity of our discussion, we assume that n and m represent the size of the two tuples.

5. CONCLUSION

Many real world applications in ubiquitous environments have to handle data with space and time dimensions that heavily require database supports. Because of this, spatiotemporal databases have gained growing attention in database communities and can give many benefits to applications closely related to space and time dimensions in ubiquitous environments.

In this paper, we have introduced three spatiotemporal data models and their query languages, which are extended from temporal data models. In order to understand which temporal data model is more user-friendly extendable to spatiotemporal data ones, we have compared the query languages using Guting's use case because query languages are tightly coupled with their underlying data models.

By evaluating the user-friendliness of the three query languages, we have found that ParaSQL is less complex than the others. For most queries, ParaSQL requires simple relation scans while SQL^{ST} and STSQL frequently need joins. Since spatiotemporal data are far more complex than ordinary data, join operations significantly degrade the system performance of database systems. In addition to this, ParaSQL expresses Query 4 with a nested form, avoiding joins. In the example, we have noted that the inner and outer queries are independent. Therefore, ParaSQL asymptotically performs this type of query with constant time complexity.

Our comparisons may not be fully sufficient to conclude that the data model of ParaSQL is superior to those of SQL^{ST} and STSQL in all aspects of spatiotemporal data. However, space and time dimensions tend to be intertwined in spatiotemporal context so that separating space and time dimensions is contrary to their characteristics, which not only causes complex query languages, but also increases implementation complexities. As we have noted, the data model of ParaSQL intertwines time and space while the others separate them. Throughout our comparisons, we can conclude that the temporal element-based data model is more legitimate than the SQL^{ST} and the STSQL for Guting's use case. Thus, it will be generally true if spatiotemporal data are similar to Guting's use case.

Finally, we want to remark on the practical implementation of the temporal element-based data model. Since a temporal element is not fixed and associated with attributes, it is not an easy task to utilize conventional database systems [24]. However, such an implementation challenge can be overcome by adopting flexible data structures. From this viewpoint, XML can be a good option for the data model [25-27]. We can directly utilize native XML database systems or build XML-based storage technologies for the data model.

REFERENCES

[1] Tamas Abraham and John F. Roddick. “Survey of spatio-temporal databases”, *GeoInformatica*, 3(1):61-99, 1999.

[2] Akio Sashima, Yutaka Inoue, and Koichi Kurumatani. *“Spatio-temporal sensor data management for context-aware services: designing sensor-event driven service coordination middleware.” In Proceedings of the 1st international workshop on Advanced data processing in ubiquitous computing*, 2006, pp.4-9.

[3] Martin Erwig, Ralf Hartmut G¨uting, Markus Schneider, and Michalis Vazirgiannis. “Spatiotemporal data types: An approach to modeling and querying moving objects in databases.” *GeoInformatica,* 3(3):269-296, 1999.

[4] John F. Roddick and Myra Spiliopoulou. “A bibliography of temporal, spatial and spatio-temporal data mining research.” *SIGKDD Explorations*, 1(1):34-38, 1999.

[5] Seo-Young Noh and Shashi K. Gadia. *“User friendly extendibility of two temporal data models to spatiotemporal data models.” In 2007 International Conference on Multimedia and Ubiquitous Engineering*, 2007, pp.241-246.

[6] David Toman. *“Point-based temporal extension of temporal SQL.” In Proceedings of the 5th International Conference on Deductive and Object-Oriented Databases*, 1997, pp.103-121.

[7] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[8] Nikos A. Lorentzos and Yannis G. Mitsopoulos. “SQL extension for interval data.” *IEEE Transactions Knowledge on Data Engineering*, 9(3):480-499, 1997.

[9] Shashi K. Gadia and Sunil S. Nair. “Temporal databases: A prelude to parametric data.” *In Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993*,* pp.28-66.

[10] Abdullah Uz Tansel and Erkan Tin. “The expressive power of temporal relational query languages.” *IEEE Transactions on Knowledge and Data Engineering*, 9(1):120-134, 1997.

[11] Jan Chomicki. *“Temporal query languages: A survey.” In International Conference on Temporal Logic*, 1994, pp.506-534.

[12] Paolo Terenziani and Richard T. Snodgrass. “Reconciling point-based and interval-based semantics in temporal relational databases: A treatment of the telic/atelic distinction.” *IEEE Transactions on Knowledge and Data Engineering*, 16(5):540-551, May 2004.

[13] Cindy Xinmin Chen and Carlo Zaniolo. *“SQL ST: A spatio-temporal data model and query language.” In Proceedings of the 19th International Conference on Conceptual Modeling*, 2000, pp.96-111.

[14] Ralf Hartmut Guting, Michael H. B¨ohlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. “A foundation for representing and querying moving objects.” *ACM Transactions on Database Systems*, 25(1):1-42, 2000.

[15] Tsz S. Cheng and Shashi K. Gadia. *“A pattern matching language for spatio-temporal databases.” In Proceedings of the 3rd International Conference on Information and Knowledge Management*, 1994, pp.288-295.

[16] S. K. Gadia and V. Chopra. “A relational model and SQL-like query language for spatial databases.” *In Advanced Database Systems*, volume 759 of Lecture Notes in Computer Science. Springer, 1993, pp.213-225.

[17] Richard J. Pasch, Daniel P. Brown, and Eric S. Blake. “Tropical Cyclone Report: Hurricane Charley.” http://www.nhc.noaa.gov/2004charley.shtml, May 2007.

[18] Michael F. Worboys. "A unified model for spatial and temporal information." *The Computer Journal*, 37(1):36-34, 1994.

[19] C. X. Chen, J. Kong, and C. Zaniolo. *"Design and implementation of a temporal extension of SQL." In Proceedings of the 19th International Conference on Data Engineering*, Bangalore, India, 2003, pp.689-691.

[20] M. R. Garey, David S. Johnson, Franco P. Preparata, and Robert Endre Tarjan. "Triangulating a simple polygon." *Information Processing Letters*, 7(4):175-179, 1978.

[21] Robert Laurini and Derek Thompson, editors. *Fundamentals of Spatial Information Systems*. Academic Press, 1992.

[22] William W. Hargrove. "Simulating Fire Patterns in Heterogeneous Landscapes." http://research.esd.ornl.gov/EMBYR/embyr.html, May 2007.

[23] *National Park Service*. "Reference/links." http://www.nps.gov/yell/technical/fire/referenc.htm, May 2007.

[24] Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. The tsql2 data model. In The TSQL2 Temporal Query Language. Kluwer, 1995, pp.153-238.

[25] F. Wang and C. Zaniolo. *"XBit: An XML-based bitemporal data model." In Proceedings of the 23rd International Conference on Conceptual Modeling*, pages 810-824, Shanghai, China, 2004.

[26] Seo-Young Noh and Shashi K. Gadia. *"An XML-based framework for temporal database implementation." In Proceedings of the 12th International Symposium on Temporal Representation and Reasoning*, Burlington, Vermont, USA, 2005, pp.180-182.

[27] Seo-Young Noh and Shashi K. Gadia. "A comparison of two approaches to utilizing xml in parametric databases for temporal data." *Information & Software Technology*, 48(9):807-819, 2006.

Seo-Young Noh

He is a senior research scientist in the Supercomputing Center of the Korea Institute of Science and Technology Information. Before joining the Institute, he worked for LG Electronics in the fields of embedded database systems and mobile linux platforms. He received his B.E and M.E in Computer Engineering from Chungbuk National University and his M.S. and Ph.D. in Computer Science from Iowa State University, respectively. His research interests are including databases, scientific data storages, information systems, distributed large-scale data handling systems, mobile linux platforms, and natural language processing.

Shashi K. Gadia

He is an Associate Professor in the Computer Science Department at Iowa State University in the U.S. and also an adjunct professor of the Birla Institute of Technology and Science in India. His research interests are including database models, query languages, incomplete information, and query optimization in temporal, spatial, and multilevel security databases.