
Toward Extensible Spatio-Temporal Databases: an Approach Based on User-Defined Aggregates

Cindy X. Chen¹, Haixun Wang², and Carlo Zaniolo³

¹ Department of Computer Science, University of Massachusetts, Lowell, MA
01854, U.S.A. cchen@cs.uml.edu

² IBM T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY
10598, U.S.A. haixun@us.ibm.com

³ Department of Computer Science, University of California, Los Angeles, CA
90095, U.S.A. zaniolo@cs.ucla.edu

Summary. User Defined Aggregates (UDAs) and table functions defined in SQL itself, rather than in an external programming language, provide a powerful mechanism for extending database languages and systems. These extensions find many applications, including data mining and spatio-temporal applications. In this paper we describe an approach where the functions and spatio-temporal primitives needed in geo-information systems can be supported by extending a small set of built-in primitives, via UDAs. This approach can be used to customize and extend spatio-temporal database systems in a variety of applications. We demonstrate the effectiveness of this approach, by implementing SQLST, a spatio-temporal data model and query language that uses counterclockwise directed triangles to model spatio-temporal objects. Our study shows that our approach achieves satisfactory overall performance.

1 Introduction

The need for spatial, temporal, and spatio-temporal databases is ubiquitous and touches several application domains, including geographical information systems, autonomous navigation, tracking, medical imaging, and many others. To answer this need, database researchers have proposed many approaches that cover a wide spectrum of data models and query languages: a very incomplete list include [7, 15, 3, 11]. The number and variety of alternative solutions proposed reflect the complexity of technical problems and the diversity of applications and requirements encountered in the field, whereby a solution proposed for a certain spatio-temporal application might not satisfy the requirements of another. To solve this problem, we propose an approach whereby the primitives built in the system can be further customized and ex-

tended by the end-users via the query language itself—native extensibility—rather than the writing and importation of foreign functions defined in an external procedural language.

Extensibility represents a long-sought-after but hard-to-attain goal for database systems. Relational DBMSs provided no extensibility whatsoever, and even in the latest generation of O-R (Object-Relational) systems extensibility comes with many limitations and requires significant expertise and programming efforts. Indeed, the main extensibility mechanism of O-R systems is to allow SQL queries to call external functions coded in a procedural language (such as C/C++ or Java). Function libraries for specific application domains are now marketed aggressively by vendors under different brand names, such as datablades, DB extenders, cartridges, or snapins. However, they all share the same severe limitations with respect to power and flexibility, inasmuch as they can only support a predefined set of operators on single records: e.g., the functions cannot access the database tables either directly or through embedded SQL calls. Furthermore, procedural attachments to SQL, such as UDFs (User Defined Functions) or stored procedures, are notorious for being inordinately hard to develop and debug [4]. Thus current datablades are not conducive to end-user extensibility (even if their source code were available, which is seldom the case).

A breakthrough in extensibility was recently achieved by the ATLaS system [23, 24, 2]. ATLaS supports the standard SQL (on top of Berkeley DB [21]), and also supports calls to functions written in procedural languages (as O-R do). But, in addition to these, ATLaS allows end-users to define new table functions and aggregates *by programming them in SQL*.

Based on ATLaS, we pursued a new approach to spatio-temporal extensions, by

- starting with a minimal set of spatio-temporal constructs as built-in C++ functions, and
- using user defined new aggregates that implement more powerful spatio-temporal operators, designed for the specific application domain.

This chapter is organized as follows. In the next section, we describe the system ATLaS developed at UCLA which deals with UDAs. In Section 3, we introduce the data model and query language of SQLST [6] by sample queries. In Section 4, we demonstrate how to use ATLaS UDAs to support spatio-temporal queries. The performance of SQLST is analyzed and discussed in Section 5. In Section 6 we discuss how to support more abstract representations and alternative ways to express SQLST queries at that level. Section 7 concludes the chapter.

2 ATLaS

ATLaS (Aggregate Table Language and System) [23, 24, 2] implements the SQL language on top of Berkeley DB record manager [21]. But in addition to supporting SQL, and allowing the users to introduce new functions by programming them in C/C++ (as O-R systems do), ATLaS supports the definition of powerful user-defined aggregates (UDAs) expressed in a SQL-like language. By programming new UDAs in SQL, end users can easily extend the database system and support a variety of advanced database applications [23]. Furthermore, ATLaS is efficient. It comes with a very limited overhead over a direct implementation in a procedural language such as C/C++ (see performance results in Section 5).

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

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

Example 1. `MOVING_AVERAGE` – a user-defined aggregate in ATLaS.

```

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

```

The first line of this aggregate function declares a local table, `STATE`, to keep (in memory) the sum and count of the values processed so far. While, for this particular example, `STATE` contains only one tuple, it is in fact a table that can be queried and updated using SQL statements. These SQL statements are grouped into the three blocks labeled `INITIALIZE`, `ITERATE` and `TERMINATE`, respectively. Thus, the `INITIALIZE` statements insert the

value taken from the input stream and sets the count to 1. The `ITERATE` statements update the table by adding the new input value to the sum and 1 to the count. The `TERMINATE` statements return the final result of computation by appending it to `RETURN`; for conformity with SQL, `RETURN` is viewed as a table which can have multiple values, and thus an `INSERT INTO` construct is used. We also add intermediate results from the computation to the `RETURN` table as part of the `ITERATE` statements.

Let us now consider the well known problem of coalescing after projection in a temporal table. We define the aggregate `coalesce`, which takes two parameters: `from` is the start time, `to` is the end time. Under the assumption that tuples are sorted by increasing start time, we can perform the task in one scan of the data. In the `ITERATE` routine, when the new interval overlaps the current interval stored in the state table, we coalesce the two intervals into one which ends with the later of the end time. Otherwise, the current interval is inserted into the `RETURN` table while the new interval becomes the current one.

Example 2. Coalescing

```

AGGREGATE COALESCE(from TIME, to TIME) :
    (start TIME, end TIME)
{
    TABLE state(cFrom TIME, cTo TIME);
    INITIALIZE : {
        INSERT INTO state VALUES(from, to);
    }
    ITERATE : {
        UPDATE state SET cTo = to
            WHERE cTo >= from AND cTo < to;
        INSERT INTO return SELECT cFrom, cTo
            FROM state WHERE cTo < from;
        UPDATE state SET cFrom = from, cTo = to
            WHERE cTo < from;
    }
    TERMINATE : {
        INSERT INTO return SELECT cFrom, cTo FROM state;
    }
}

```

The fact that UDAs in ATLaS are written in SQL achieves compatibility of data types and programming paradigms and inherits the well-known benefits of database query languages, such as scalability, data independence and plausibility. The ability of introducing and manipulating new tables is one of the first cornerstones of ATLaS' power. The second is the ability of one aggregate calling another, or calling itself recursively.

The ATLaS compiler translates ATLaS programs into C++ code. ATLaS adopts an open interface for its physical data model, so that the system can link with a variety of physical database implementations. The Berkeley DB

library [21] is now used as ATLaS' main storage manager, but we have now added support for in-memory tables, and there is current work to support R-trees [16].

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

3 SQLST

In this section, we give a conceptual level description of SQLST [6], which was designed following the general framework proposed by Worboys [26], where a spatio-temporal entity is modeled as a unified object having orthogonal spatial and temporal extents. Furthermore, Worboys' approach based on simplexes by representing 2-simplexes (triangular areas) [25] was improved by *counterclockwise directed triangles*. The virtue of counterclockwise directed triangles lies in the simple solution to the point-location problem [19], i.e., testing whether a point is *inside* a polygon, where a polygon is represented as set of triangles. The point-location problem is an important computational geometry problem and it is the basis to determine spatial relationships between spatial objects.

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

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

Given a counterclockwise directed triangle T and an arbitrary point P, shown in Figure 1, below,

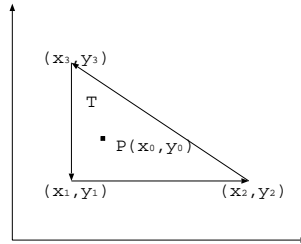


Fig. 1. An example of counterclockwise directed triangle

the determination of whether P is inside the triangle is performed by the predicate function

inside(point P, triangle T) *which is true if the property*

$$\begin{vmatrix} x_i & y_i & 1 \\ x_j & y_j & 1 \\ x_0 & y_0 & 1 \end{vmatrix} \geq 0$$

holds for: (i, j) = (1, 3), (2, 3), and (3, 1).

If a point $P = (x_0, y_0)$ is on the *left* of all the directed edges of the triangle $T = ((x_1, y_1), (x_2, y_2), (x_3, y_3))^4$, then P is *inside* T .

Our basic spatial objects are *points*, *lines (finite line segments)*, and *counterclockwise directed triangles*; thus, a polygon is represented by a set of counterclockwise directed triangles. For example, the Hawaii island Maui is represented by more than 100 adjacent triangles with vertices specified in geographic coordinates, i.e., longitude and latitude.

By decomposing polygons into sets of triangles, hard-to-express spatial relationships between two spatial objects can be evaluated easily [9, 25]. The algorithms to decompose a polygon into triangles and to merge triangles into polygons can be found in [20, 18].

The SQLST system supports the following built-in operators:

- **perimeter**
- **area**
- **inside**
- **distance**
- **intersect**
- **equal**
- **overlap**
- **contain**
- **disjoint**
- **adjacent**

The first two operators are unary operators that take as argument a polygon, and the others are binary operators. The **inside** operator takes as first operand a point and as a second operand a polygon. All the remaining operators take as operand two polygons.

We now introduce SQLST through examples that illustrate its use in supporting spatio-temporal queries posed on cyclone-tracking database at a weather center. Specifically, we consider a spatio-temporal application from the NSF Arctic System Science (ARCSS) Research Program's Ocean Atmosphere Ice Interactions (OAI) Project. The marine environment is an interactive system comprising the water, ice, air, biota, dissolved chemicals and sediments. ARCSS/OAI seeks to enhance understanding of this system and its role in climate and global change. The extratropical cyclone data set [1] at ARCSS provides information about cyclones in the Northern Hemisphere.

⁴ A point is considered to be on the left of a line if it is on the (extended) line.

We will use this data set throughout our study. Included in this database is information about cyclone activities such as the cyclones' trajectories and the pressure along the trajectories during certain time intervals.

Example 3. Define the Cyclone relation.

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

In this relation, the Tstart and Tend columns indicate the start and end instants of a time interval, and have a granularity of one day while the granularity could be larger or smaller, such as one month or one hour. Trajectory is a line segment whose start and end points are the positions (longitude, latitude) of the cyclone at time Tstart and Tend, respectively. Pressure is the average pressure between Tstart and Tend, and its unit is *milibar*. ID is the cyclone identifier. In this test case, the movement of cyclones are measured every day, so ID and Tstart together serve as the key of the relation.

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

Example 4. Define the Island relation.

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

In this relation, the Region column stores the geometry of the island. Each polygon is decomposed into non-overlapping triangles. For example, our table contains about 100 rows each with Name = "Maui".

Following is a set of typical cyclone-related queries for storm tracks data set. We have also used similar queries to analyze the computer simulation data produced by a climatic weather model on another NASA-sponsored project [17].

Example 5. Find the cyclones which have pressure greater than 1000mb for more than three days during their life cycle.

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

This query will return the IDs of the cyclones whose overall high pressure stage have lasted more than three days.

As it can be easily inferred from the syntactic structure of this query, the GROUP BY and HAVING constructs, the operator DURATION is in fact a user-defined aggregate. The definition and implementation in SQLST of this aggregate, as well as others that appear in this section will be discussed in detail in Section 4.

Example 6. Find the cyclones whose trajectory was enclosed by Maui.

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

This query returns the ID of the cyclones that have been completely inside the island "Maui". Aggregates in ATLaS can return any number of results. For instance, the aggregate `CONTAIN(Region, Trajectory)` returns a result only if the trajectory is completely inside the region. Thus `CONTAIN(Region, Trajectory)` in this query corresponds to the boolean condition that some result is returned.

Example 7. Find how long each cyclone has traveled when it was over Maui.

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

This query returns the total distance the cyclones traveled when their trajectories overlapped with or were contained by the region of the island "Maui".

The functions `intersect` and `length` are built-in C++ functions. When a line (`Trajectory`) intersects a triangle (`Region`), the intersection is a line or a point. The length of these lines is then calculated and summed up.

Other spatial relationships similar to `CONTAIN` and `OVERLAP` include `ADJACENT`, `DISJOINT`, etc. The implementation of these operators is discussed in the next section.

Example 8. Identify all cyclones that have come within 50 miles of the coast of Lanai.

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

This query finds the cyclones that have come within 50 miles to the coast of the island "Lanai" in their trajectories .

Example 9. Find the cyclones that have traveled more than 300 miles continuously.

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

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

In the above examples, we find SQLST is a natural minimalist extension to SQL. Besides operators such as DURATION, CONTAIN, OVERLAP, EDGE_DISTANCE and MOVING_DISTANCE, users can define their own operators such as ADJACENT, DISJOINT, etc., also as user-defined aggregates. We will explain more about user-defined aggregates in the next section.

4 Spatio-Temporal Operators in ATLaS

Many abstractions have been proposed to deal with the complexity and diversity of temporal reasoning and spatial objects encountered in reality. Here we study how spatio-temporal operators used in the queries of the previous section can be expressed using (user-defined aggregates (UDAs)); we first discuss temporal operators, then spatial ones, and finally the two combined.

4.1 Temporal Aggregates

We implemented our SQLST system by first adding the built-in functions into ATLaS, and then using its UDA mechanism to implement the spatio-temporal aggregates needed in spatio-temporal queries. For instance, the temporal aggregate DURATION in Example 5 is implemented as follows:

Example 10. DURATION

```

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

```

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

```

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

```

The result of DURATION will be 15 days.

4.2 Spatial Aggregates

UDAs can be used to implement the operators discussed in Section 3 on general polygons represented as sets of triangles, using the built-in operators on individual triangles. The implementation of `EDGE_DISTANCE`, `CONTAIN` and `OVERLAP` is discussed next.

The `EDGE_DISTANCE` between two polygons, each represented as a set of triangles, is the smallest distance between the vertices of triangles in either polygon and the edges of triangles in the other.

Example 11. `EDGE_DISTANCE`

```

AGGREGATE EDGE_DISTANCE(Object1 TRIANGLE,
                        Object2 TRIANGLE) : REAL
{
  TABLE state(d REAL);
  INITIALIZE : ITERATE : {
    INSERT INTO state
      SELECT distance(V,E)
      FROM TABLE(vertex(Object1)) AS V,
           TABLE(edge(Object2)) AS E;
    INSERT INTO state
      SELECT distance(V,E)
      FROM TABLE(vertex(Object2)) AS V,
           TABLE(edge(Object1)) AS E;
  }
  TERMINATE : {
    INSERT INTO return SELECT MIN(d) FROM state;
  }
}

```

In this example, we use the table functions `vertex` and `edge` which take as input an object of type `triangle`, and return its vertices and sides, respectively. Table functions are a very useful SQL:1999 [10] construct supported in most commercial DBMSs. ATLaS also supports table functions, where they can either be written in an external procedural language, or in SQL itself, as in the case of UDAs. The function `distance`, used in Example 11, is instead a standard scalar function that calculates the Euclidean distance between a point and a line segment.

We next discuss the implementation of `CONTAIN`. Here we take advantage of inheritance and overloading characteristics of O-R systems [22]. Both point and line are subtypes of `triangle`. A point can be considered as a triangle whose three vertices are the same and a line as a triangle whose first two vertices are the start and end points of the line and the last vertex is the center of the line.

Example 12. CONTAIN

```

AGGREGATE CONTAIN(Object1 TRIANGLE, Object2 TRIANGLE)
: INT
{
  TABLE state(b INT);
  TABLE triangles(Object TRIANGLE);
  TABLE points(Vertex POINT);
  TABLE edges1(line1 LINE, count1 INT);
  TABLE edges2(line2 LINE, count2 INT);
  INITIALIZE : {
    INSERT INTO state VALUES 1;
  }
  ITERATE : {
    INSERT INTO triangle VALUES(Object1);
    INSERT INTO points VALUES(Object2.Vertex);
    INSERT INTO edges1 VALUES(Object1.Edge);
    UPDATE count1 = count1 + 1 WHERE Object1.Edge IN
      (SELECT line1 FROM edges1);
    INSERT INTO edges2 VALUES(Object2.Edge);
    UPDATE count2 = count2 + 1 WHERE Object2.Edge IN
      (SELECT line2 FROM edges2);
  }
  TERMINATE : {
    UPDATE state SET b = 0 WHERE NOT EXIST
      (SELECT Vertex FROM points, triangles
        WHERE inside(Vertex, Object) = 1);
    OR EXIST
      (SELECT intersect(line1,line2) FROM edges1, edges2
        WHERE count1 = 1 AND count2 = 1);
    INSERT INTO return SELECT b FROM state WHERE b = 1;
  }
}

```

The function *inside* is a built-in function that will evaluate to 1 if a point is *inside* a triangle; 0 otherwise; and the function *intersect* is a built-in function that will return the common part of two intersect lines. It does not return anything if the lines do not overlap or cross.

The spatial aggregate `CONTAIN(Object1, Object2)` then uses *inside* and *intersect* to test if Object₁ contains Object₂. The aggregate first iterate through all the records and insert the triangles belonging to Object₁ into one table, the vertices of Object₂ into another table, and their edges into two other different tables. The edges are associated with a count. If the count is 1, then the edge only occurs once in the set of triangles; thus the edge is an outer edge, i.e., an edge of the original polygon.

Initially, *b* in the `STATE` is set to 1. If there exists a vertex of Object₂ that is not contained in any triangle of Object₁, or if their outer edges cross, then *b* will be set to 0, thus not inserted into the `RETURN` table. As a result, the

aggregate `CONTAIN` will not return anything; otherwise, it will return 1, the value of `b`.

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

Example 13. `OVERLAP`

```

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

```

Similar to `CONTAIN`, the spatial aggregate `OVERLAP` uses `intersect` to test if an edge of a triangle belonging to `Object1` crosses an edge of a triangle belonging to `Object2`. Therefore, the aggregate first iterates through all the records and inserts the edges of `Object1` and `Object2` into two different tables. If there exists two crossing edges, then the value of `b` will be set to 1, which will then be returned by the aggregate `OVERLAP(Object1, Object2)`; otherwise, `OVERLAP(Object1, Object2)` will not return anything.

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

4.3 Spatio-Temporal Aggregates

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

Example 14. `MOVING_DISTANCE`

```

AGGREGATE MOVING_DISTANCE(Object, Tstart DATE,
Tend DATE) : REAL
{
  TABLE state(d REAL, x REAL, y REAL, time DATE);

```

```

INITIALIZE : {
  INSERT INTO state
    VALUES(0, centerx(Object), centery(Object), Tend);
}
ITERATE : {
  UPDATE state SET
    d = d + sqrt((centerx(Object) - x)2 + (centery(Object) - y)2),
    x = centerx(Object), y = centery(Object), time = Tend
  WHERE Tstart = time + 1;
  INSERT INTO state
    VALUES(0, centerx(Object), centery(Object), Tend);
  WHERE Tstart ≠ time + 1;
}
TERMINATE : {
  INSERT INTO return SELECT d FROM state;
}
}

```

This aggregate calculates the distance an object travels in each continuous time period. We calculate and sum up the distance between two positions of the center of an object at two consecutive time intervals. When the start of a new time interval is not consecutive to the ends of any existing time intervals in the `state` table, we start a new round of calculation. Upon termination, all the `d` values in the `state` table, i.e., the distance traveled for each continuous time period, are returned. The built-in functions `centerx` and `centery` calculates the x and y coordinates of the center of an object, respectively.

5 Performance

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

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

Example 15. Find the duration of the cyclones occurred in June, 1966.

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

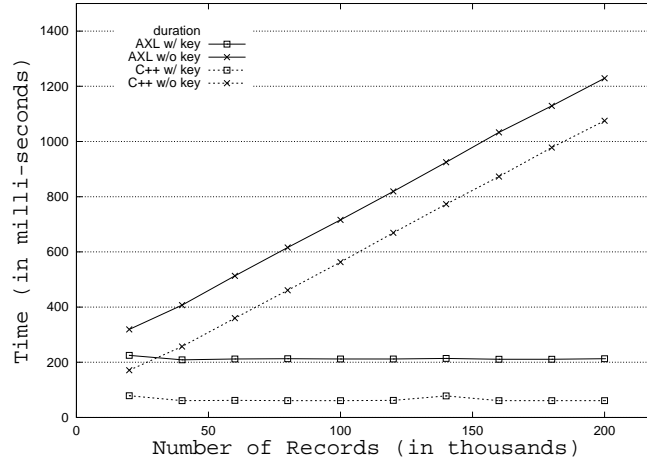


Fig. 2. Performance result of Query 15

Example 16. Find the distance traveled by the cyclones occurred in June, 1966.

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

Figures 2 and 3 are the results of the aggregates DURATION and MOVING_DISTANCE. Since the number of records that meet the selection criterion is fixed, so when a key is used, we only need to retrieve the qualified records, thus the result of performance of ATLaS and C++ are constant. When no key is used, we need to search the entire database, so the result of the performance of both ATLaS and C++ increase linearly. In both cases, the program written directly in C++ against the Berkeley DB API outperforms ATLaS by a constant.

Example 17. Find the cyclones which occurred in June, 1966 and landed on the island Oahu.

```
SELECT ID
```

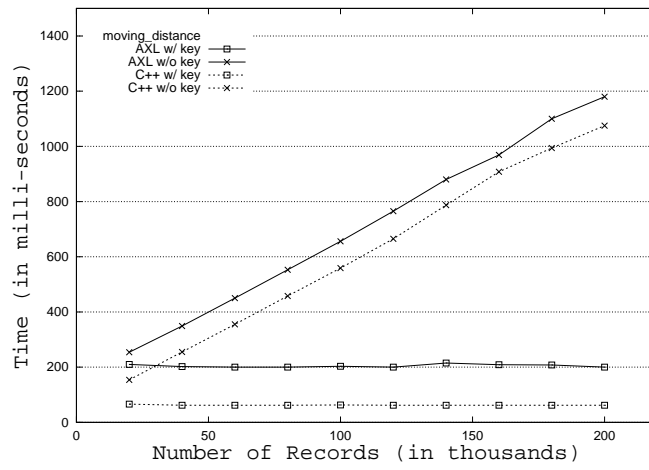


Fig. 3. Performance result of Query 16

```

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

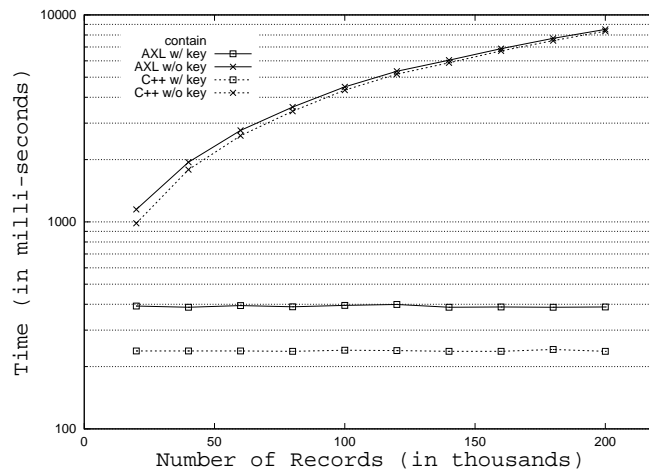


Fig. 4. Performance result of Query 17

Example 18. Find the distance between cyclones which occurred in June, 1966 and the coast of island Oahu.

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

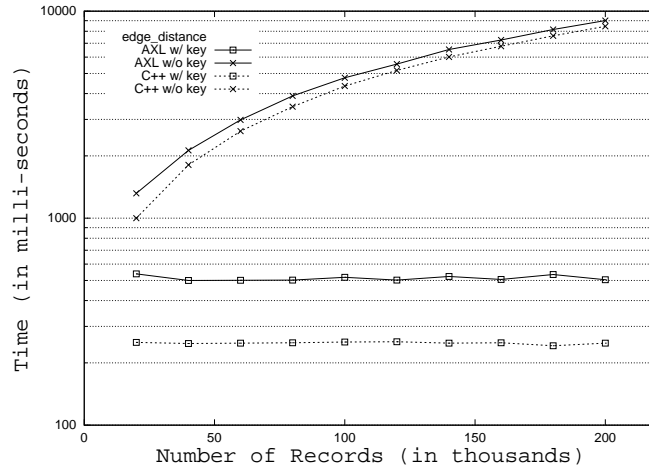


Fig. 5. Performance result of Query 18

Figure 4 and 5 show, in logarithmic scale, the results of the aggregates `CONTAIN` and `EDGE_DISTANCE`. Still, when a key is used, the results of performance of ATLaS and C++ are constant. When no key is used, the results of the performance of both ATLaS and C++ increase. Again, In both cases, C++ API of Berkeley-DB is slightly faster than ATLaS. Unlike the aggregates `DURATION` and `MOVING_DISTANCE` which handle the incoming data as soon as it comes in without storing it, the aggregates `CONTAIN` and `EDGE_DISTANCE` need to insert the incoming data into temporary tables first, and then upon termination, query these tables to get the desired results. Thus the performance curve of these two aggregates is different from that of `DURATION` and `MOVING_DISTANCE`.

The performance results show that the user-defined aggregates of SQL^{ST} only incur in a modest performance penalty for the ease of use and extensibility they provide.

6 Toward More Abstract Representations

The extensibility mechanisms of SQLST can also be used to elevate the representation to richer and more abstract levels. For instance, at the conceptual level, polygons provide a powerful representation for spatial objects. In terms of temporal representation, users may want to see the movement of the cyclones as a series of snapshot of their positions. In this section, we show how SQLST can be further extended to support these more abstract representations. In fact, we will now express the queries in Section 3 using *points* and *polygons* as spatial data types and *time instants* as temporal data type.

6.1 Schema Definition

The schema of the databases in Section 3 have been changed as follows.

Example 19. Define the Cyclone relation. (Example 3)

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

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

Example 20. Define the Island relation. (Example 4)

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

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

6.2 Spatio-Temporal Queries

Since the spatio-temporal operators are supported at the internal level, so table expressions in O-R systems [4] are used to transform the queries.

For instance, Example 5 in Section 3 is expressed as follows at the conceptual level.

Example 21. Find the cyclones which have pressure greater than 1000mb for more than three days during their life cycle (Example 5).

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

In this query, we use table expressions in the FROM clause to accomplish the transformation between different levels of abstraction.

To map a pairs of points and time instants into a line and a time interval, we define the UDA MAPPING:

Example 22. MAPPING

```

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

```

Similarly, Example 6 can be expressed as follows.

Example 23. Find the cyclones whose trajectory was enclosed by Maui (Example 6).

```

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

```

Decompose is a built-in function which triangulates a polygon (Extent in this query) into a set of triangles (Region). The algorithm of decompose can be found in [20, 18].

7 Conclusions

In this chapter, we described a novel approach to extensible data models and query languages to manage spatio-temporal information, and demonstrated its

benefits in implementing the SQLST system. By building on the native extensibility mechanisms of ATLaS, we minimized the external procedural extensions required, since only user-defined aggregates and table functions defined in SQL are now needed to support powerful spatio-temporal constructs.

In SQLST, end-users can easily introduce their own spatio-temporal operators, and retain the flexibility of working at different levels of abstraction as best fit their needs. Current work focuses on adding new spatial and temporal indices, including R-trees [16], and the optimization of complex queries with temporal joins and spatial joins.

References

1. ARCSS Data and Information Archive Cyclone Track Data Set. <http://arcss.colorado.edu/Catalog/arcss003.html>
2. ATLaS User Manual. <http://wis.cs.ucla.edu/atlas>
3. M. Cai, D. Keshwani, and P. Z. Revesz. Parametric Rectangles: A Model for Querying and Animation of Spatiotemporal Databases. In *Proceedings of the 7th International Conference on Extending Database Technology*, pp.430-444, 2000
4. D. Chamberlin. *A Complete Guide to DB2 Universal Database*, Morgan Kaufmann, 1998
5. C. X. Chen and C. Zaniolo. Universal Temporal Extensions for Data Languages. In *Proceedings of the 15th International Conference on Data Engineering*, pp.428-437, 1999
6. C. X. Chen and C. Zaniolo. SQLST: A Spatio-Temporal Data Model and Query Language. In *Proceedings of the 19th International Conference on Conceptual Modeling*, pp.96-111, 2000
7. J. Chomicki and P. Z. Revesz. Constraint-Based Interoperability of Spatiotemporal Databases. In *Advances in Spatial Databases*, LNCS 1262, pp.142-161, Springer, 1997
8. K. L. Clarkson, R. E. Tarjan and C. J. Van Wyk. A Fast Las Vegas Algorithm for Triangulating a Simple Polygon. In *Discrete & Computational Geometry*, Vol.4, No.5, pp.423-432, 1989
9. M. J. Egenhofer, A. U. Frank and J. P. Jackson. A Topological Data Model for Spatial Databases. In *Symposium on the Design and Implementation of Large Spatial Databases*, pp.271-286, 1989
10. A. Eisenberg and J. Melton. SQL: 1999, formerly known as SQL 3. In *SIGMOD Record*, Vol.28, No.1, pp.131-138, 1999
11. L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp.319-330, 2000
12. A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. In *ACM Transactions on Graphics*, Vol.3, No.2, pp.153-174, 1984
13. R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. In *ACM Transactions on Database Systems*, Vol.25, No.1, pp.1-42, 2000
14. M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a Simple Polygon. In *Information Processing Letters*, Vol.7, No.4, pp.175-179, 1978

15. S. Grumbach, P. Rigaux and L. Segoufin. The DEDALE System for Complex Spatial Queries. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp.213-224, 1998
16. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp.47-57, 1984
17. R. Muntz, E. Shek and C. Zaniolo. Using *LDL++* For Spatio-Temporal Reasoning in Atmospheric Science Databases. In *Applications of Logic Databases (R. Ramakrishnan, eds.)*, pp. 101-119, 1995
18. J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1998
19. F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer, 1988
20. R. Seidel. A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons. In *Computational Geometry: Theory and Applications*, Vol.1, No.1, pp.51-64, 1991
21. Sleepycat Software. The Berkeley Database (Berkeley DB). <http://www.sleepycat.com>
22. M. Stonebraker, P. Brown and D. Moore. *Object-relational DBMSs: tracking the next great wave*, Morgan Kaufmann, 1999
23. H. Wang and C. Zaniolo. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. In *Proceedings of the 26th International Conference on Very Large Databases*, pp.166-175, 2000
24. H. Wang and C. Zaniolo. ATLaS: A Native Extension of SQL for Data Mining. In *Proceedings of the 2nd SIAM International Conference on Data Mining*, 2003
25. M. F. Worboys. A Generic Model for Planar Geographical Objects. *International Journal of Geographic Information Systems*, Vol.6, No.5, pp.353-372, 1992
26. M. F. Worboys. A Unified Model for Spatial and Temporal Information. *Computer Journal*, Vol.37, No.1, pp.26-34, 1994