Design of Spatial Query Language for GEO Millennium Server™

*Zhaohong Liu, Sung-Hee Kim, Young-Hwan Oh, Hae-young Bae
Dept. of Computer Sci. & Eng., Inha Univ., Inchon, 402-751, Korea
*macromliu@hotmail.com

Abstract

A GIS software GEO Millennium System ™ has been developed to integrated with spatial database that combines conventional and spatially related data. As we known well the standard query language lacks the support of spatial data type and predicate, and can not serve as the query language in the spatial database directly; some extended strategies have been proposed, but some of them need their own storage manager, some introduce new clause into the SELECT-FROM-WHERE structure, and some is very complex and available to us. So we designed our own query language on the conventional storage manager system. It supports the Spatial Data Type and predicate, and provides the full query capabilities of SQL on the non-spatial part of the database while being tightly integrated with the spatial part, without changing the standard SQL structure.

1. Introduction

Data intensive geographic applications such as cartography, urban planning, and natural resource management are build by GISs. GISs are database systems that allow the manipulation, storage, and retrieval of geographic data and the display of data in the form of maps. The spatial DBMS provides the underlying database technology for GISs and other applications. Spatial DBMS (i) is a database system; (ii) offers spatial data types (SDT) in its data model and query language; (iii) supports spatial data types in its implementation, providing at least spatial indexing and efficient algorithms for spatial join. [7]

We adopted a conventional storage manger system to store the data. It only supports the traditional atomic data type. The relational language (SQL) forms a suitable base to develop spatial extensions, but it cannot support for spatial data type and predicate. So we must do some work to implement our own spatial query language that supports both spatial data type and spatial data manipulation.

Section 2 gives an introduction about the project "GEO Millennium System ™+". Section 3 presents the features of spatial data, the available ways, and principle for spatial query language, the advantages and drawbacks of these methods and our design of the spatial query language. Section 4 dates some conclusions.

2. Introduction of the GEO Millennium System ™

We adopted the conventional Client/Server model in the GEO Millennium System ™. It is composed with GEO Millennium Server ™ and GEO Millennium Client ™ [Figure 1].

The GEO Millennium Client ™ provides the GUI (Graphics User Interface). It is composed of Communication Module, Cache management, Local Query Processor, Drawing Module and User Interface Module.

The GEO Millennium Server ™ is the system's server program, it responses for storing and managing the spatial and non-spatial data in a DBMS, provides computing function, and serves multi-client at the same time. The GAPE (Geo-spatial API Processing Engine) provides API function for the client, translates the packet format received from user into database query, and manages the spatial data schema and user. The QPE (Query Processing Engine) parses the queries received from

+ GEO Millennium System ™, GEO Millennium Server ™ and GEO Millennium Client ™ is the registed trademark of GEOMania Co. Ltd. and DB Lab. Inha Univ. KOREA

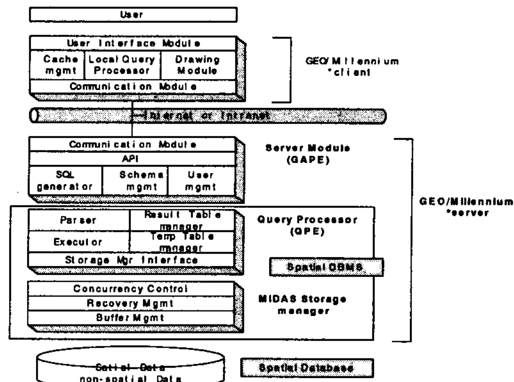

Figure 1: Architecture

GAPE, executes it using the interface supported by MiDAS-III. In the GEO Millennium Server ™, it uses the spatial query language supporting spatial operation, which is extended from basic SQL, and supports intersect, disjoint, contain, equal, touch, cross and overlap etc. spatial operators.

3. The Spatial Query Language

In this section, we firstly give a brief introduction of the GIS/Spatial DBMS, SQL; secondly, the features of the spatial data and spatial predicates; thirdly, the available way presented to solve the problem, and the advantage and shortcomings; fifth, the requirements of the extended SQL; lastly, our own spatial query language solution methods also be introduced.

The first generation of GIS was built directly on file systems and did not offer the benefits of DMBS such as high-level data definition, flexible querying, transaction management, etc. When DBMS technology and in particular, relational systems [1], in which security and backup and recovery are well proven become available, attempts were made to use them as a basis and take the advantages.

SQL was originally designed as the query for System R [1], the IBM's relational DBMS. The structure of an SQL query is the SELECT-FORM-WHERE clauses. The large availability on the market place of the relation database technology is the major reason why an SQL-based spatial query language is welcome both from the GIS vendors and the GIS user community.

The traditional alphanumeric data, such as integer, real, character and string is the simplest data, it can be easily written

in the SQL sentence. But in the spatial system, the fundamental abstractions are point, line, and region [7]. A point represents (the geometric aspect of) an object for which only its location in space, but not its extent, is relevant. A line (in this context always understood to mean a curve in space, usually represented by a polyline, a sequence of line segments) is the basic abstraction for facilities for moving through space, or connections in space (e.g., roads, rivers, cable for phone, electricity). A region is the abstraction for something having an extent in 2-D space (e.g., country, lake, or national park).

Some important issues related to spatial data types are following:

- Extensibility
- Completeness
- One or more type? Is it really necessary to have several different types to distinguish? We choose several types, it allow a more precise application of operations.
- Set operations. [7]

Base on these principles, we adopt the point, line, polyline, ring, rectangle, roundrect, circle, ellipse, and polygon as the atomic spatial data type, and another one MBR (Minimum Boundary Rectangle) is also been introduced.

Among the operations offered by spatial algebras, spatial relationships are the most important. The spatial relationship can be distinguished several classes [7].

- Topological relationship, such as adjacent, inside, and disjoint
- Direction relationship, for example, above, below, or north_of, southwest_of.
- Metric relationships, for example, "distance < 100."

Eight of these are not valid, and two of them are symmetric so that six different relationships result, called disjoint, in, touch, equal, cover, and overlap. And for easy use we adopt 7 called intersect, disjoint, contain, equal, touch, cross and overlap, and another very important predicate *window* which was used in the window query.

Recently several spatial extensions to standard SQL have been proposed to turn SQL into a spatial query language In all such extensions, the motivation to adopt SQL as the backbone for a spatial query language was based on the recognition of the efforts to standardize SQL as the database query language.

Extensions to make SQL usable within Spatial DBMS must include: (i) spatial data type, (ii) spatial operators predicates, and (iii) support for both textual and graphical presentation of query results [6]. It is a very good idea, but the integration of the graphical presentation into the query language would make each user unnecessarily complex and long. In [8], the Constraint SQL is based on the constraint database. Some other extended way extended query language has been an incomplete implementation of the ANSI standard [2], and, furthermore, the spatial extensions were fairly minimal and elementary. To make matters worse, often the extensions do not maintain consistent syntactic and semantic constructs with the rest of standard SQL. For example, spatial predicates are not in general supported within the WHERE clauses, but rather within a separate clause [4].

In the context of spatial database, a query language must be seen as broader than only a solution to the retrieval of data. The standard structure of SQL with the SELECT-FROM-WHERE block is already considered to be complex enough to us. But it only well suited to treat alphanumeric data, it do not reflect the properties of spatial data. A bellowing, we list the set of basic features [4].

1.A spatial abstract data type. A spatial data type is necessary so that the users may treat spatial data at a level of abstraction independent from the internal coding.

2.Spatial selection criteria and selection by pointing. The user needs to select data to be retrieved not only based on predicates over attributes values (e.g., the standard query "select all professors older than 50 years"), but also based on spatial properties (e.g., "select all parcels within 100 meters of a lake"). Hence, a spatial query language must allow predicates to select data based on relationship of spatial objects.

Three fundamental categories of queries can be distinguished in a spatial DBMS:

- Queries about non-spatial properties; e.g., "How many people do they live in Rome?"
- Queries about spatial properties; e.g., "query all towns near by lake Ontario";
- Queries which combines spatial and non-spatial properties; e.g., "Query all the parcels adjacent to the parcel located at 25 College Street."

It is crucial, for spatial query language, to syntactically support these three categories of queries. Traditional query provides a solution for the formulation of non-spatial queries. So the spatial extension to a non-spatial language must preserve all its alphanumeric functionality to allow the user to pose non-spatial queries easily. And we do not integrate the display description into the query language, or it would make the query complex and long.

The following concepts of standard SQL were specifically regarded:

- The SELECT-FROM-WHERE construct is the framework of every query;
- Predicates in the WHERE clause are formulated upon attributes or tables (only for the spatial predicates);
- Every query result is a relation.

We adopt the Purdue Compiler Construction Tool Set (PCCTS)[9] as our parser tools. And in the following the given grammar is using the grammar.

1) Spatial Data Type Definition

In the GEO Millennium Server ™, based on the extensibility and completeness principle said before, we consider the POINT, LINE, POLYLINE, RING, RECT, ROUNDRECT, CIRCLE, ELLIPSE, and POLYGON as the atomic data type. It is very clear in the CRATE TABLE sentence.

Here is the CREATE TABLE SQL's grammar.

sqlCreateTableStmt :
  CREATE TABLE *tablename* LBRACE
  newfielditem (COMMA *newfielditem*)* RBRACE ;
newfielditem :
  newfieldname fieldType {PRIMARY KEY | INDEX KEY};
fieldType :
  ((CHAR LMBRAC *IntNum* RMBRAC)
  | VCHAR | INT | LONG | DOUBLE | MBR
  | POINT | LINE | POLYLINE | RING | RECTANGLE
  | ROUNDRECT | CIRCLE | ELLIPSE | POLYGON)

In these data type, except the MBR have a fixed length, the other spatial data type POINT, LINE, POLYLINE, RING, RECT, ROUNDRECT, CIRCLE and ELLIPSE are a set of point, and a point has the structure like (double, double), the length is variable, we adopt the LIST type supported by the storage manager to store it. The MBR that has the structure like (long, long, long, long), we store it as char [32].

2) The solution of the spatial SQL

Predicate should be provided to manipulate the data. The projection, of course, the related operator must be provided. In the parser party, the parser of the INSERT, SELECT, UPDATE, DELETE should be modified. The method deal with the WHERE clause of the SELECT, UPDADTE, DELETE.

As we have referred previously, we've supported the SDT, and the related predicates INTERSECT DISJOINT CONTAIN EQUAL TOUCH CROSS OVERLAP and WINDOW must been supported also. We do not introduce new clause as some old method, just merge all the predicates into the where clause in the SELECT-FROM-WHERE style.

Here we only give out the grammar for SELECT command.

```
sqlSelectStmt :
    SELECT (TIMES | (selectItems (COMMA selectItems)*))
    FROM (tablename (COMMA tablename )* )
    {WHERE   sqlMultiPredicate };
selectItems :
        OID| (stringidentify {PERIOD( stringidentify | OID)});
sqlMultiPredicate : sqlORExpr ( OR sqlORExpr )*;
sqlORExpr : sqlANDExpr ( AND sqlANDExpr )*;
sqlANDExpr : ( ( NOT )* sqlNOTExpr ) ;
sqlNOTExpr :
    (LBRACE sqlMultiPredicate RBRACE)
    | sqlSinglePredicate ;
sqlSinglePredicate :
    ( ( INTERSECT | DISJOINT | CONTAIN | EQUAL
      | TOUCH     | CROSS    | OVERLAP ) LBRACE
      ( (stringidentify {PERIOD LMBRAC IntNum
         COMMA IntNum COMMA IntNum  RMBRAC } )
       | ( LMBRAC IntNum  COMMA  IntNum COMMA
         IntNum  RMBRAC) )
      COMMA  stringidentify RBRACE )
    | ( (OID LBRACE IntNum COMMA IntNum
         COMMA IntNum RBRACE))
    | ( WINDOW LMBRAC  (IntNum  | RealNum )
      COMMA  ( IntNum  | RealNum )
      COMMA  ( IntNum  | RealNum )
      COMMA  ( IntNum  | RealNum ) RMBRAC  )
    | // normal predicate  and table.OID
     ( stringidentify
      ( ( (EQ  | GT | GE | LT | NE)  sqlTerm )
        | PERIOD (  ( OID LBRACE IntNum COMMA
          IntNum COMMA IntNum  RBRACE  )
           | (stingidentify (EQ | GT | GE | LT | NE ) sqlTerm ))
      ));
```

The OID (Object Identifier) has a special usage, when it is used in the *sqlSelectItems*, it means return the Object Identifier number of the record, when it is used in the following condition: the object data has been cached in the client side, we do not need retransfer the data itself from the server, it's only necessary require the OID, this strategy can reduce the transferred data size between server and client. In the where clause, it refer to a record which stored a geometry, it is different with the conventional data type, first, we read the record from the table, it contain a MBR field, and the object itself, then we use the select-refine two phase strategy to retrieve the query result.

We consider the entire spatial predicate into the *where* clause, including the *WINDOW* predicate, so we can deal with it as same as the alphanumeric predicate, and in the Normalization phrase, we adopted the Conjunction Normalization algorithm.

Conjunction Normalization Algorithm:

```
CNF_normalizer(BinaryTree* p_tree){
    if p_tree is a terminal node
      return a AND node pointing to an OR node, and attach
      the terminal node to the OR node;
    else {
      left = CNF_normalizer( p_tree's left subtree);
      right = CNF_normalizer(p_tree's right subtree);
      switch the intermediate node's type: {
      case AND:
        return CNF_and_merge(left, right); // plus
      case OR:
        return CNF_or_merge(left, right); // multiply
      };
    }
}
```

After the condition was normalized, we can retrieve it from the storage manager system using the provided interface. Of course we can optimize access path, and reduce the unnecessary Cartesian.

4. Conclusions and Future Research

In this paper, we've discussed the implemented spatial query language used in the GEO Millennium Server ™. We proposed the methods and principles for spatial query language, extended SQL, and also their advantages and shortcomings. We observe their advantages, at the same time avoiding the drawbacks such as complex to use and the syntactic and semantic violation with the standard SQL, at last, we implemented our own spatial query language used in the GEO Millennium Server ™, which has the full query capabilities of SQL on the non-spatial part of the database while being tightly integrated with the spatial part, and store the spatial data in the conventional storage manager.

About the future research, we'll support more useful functions and predicates, and apply the optimization to it.

REFERENCES

[1] M. Astrahan et al., "System R: A Relational Approach to Data Base Management", ACM TODS, pp.97-137, 1976

[2] C.J.Date, "A Guide to the SQL Standard", Addison-Wesley, Reading, Mass., 1989.

[3] M.J. Egenhofer. "Spatial SQL: A Query and Presentation Language". IEEE TKDE, 6(1): pp.86-95, 1994.

[4] M.J. Egenhofer and A.U. Frank. "Towards a Spatial Query Language: User Interface Considerations", Proc. 14th Conf. On VLDB, pp.124-133, Los Angeles, CA, Aug.1988

[5] Martin Erwing & Markus Scheider. "Development in Spatial-temporal Query Language". 1999

[6] Paolino Di Felice etc. "Towards a Standard for SQL-Based Spatial Query Language", ACM pp.184-189, 1992.

[7] Ralf Hartmut Guting, "An Introduction to Spatial Database Systems", VLDB Vol.3 No.4, pp.357-399, 1994.

[8] Gabbriel Kuper etc., "A Constraint-based Spatial Extension to SQL", ACM GIS'98 Washington, pp.112-117, 1998.

[9] Terence John Parr. "Language Translation Using PCCTS and C++ A reference guide". Automata Publishing Company, San Jose, CA 95129.

[10] N.Roussopoulos and D.Leifker, "Direct Spatial Search on Pictorial Database Using Packed R-Trees", Proc. Int. Conf. On Management of Data, SIGMOD, pp.17-31, 1985