Abstract: GEUS has been developed by extending a commercial object-relational database engine itself. Spatial query processing was tightly integrated with the aspatial query engine to support spatial primitive types, operators, and path expressions that returning the characteristic values of spatial objects. For the fast retrieval of spatial objects, GEUS has been equipped with an enhanced R*-tree as a spatial access method. Consequently, aspatial query planning and optimizing mechanism has been largely modified so that the extended query engine takes into account the query costs estimated from the selectivity of the spatial operators as well as the common database statistics.

This kind of tight integration approach makes GEUS acquire better system performance than the other conventional plug-in systems, especially in the case of handling extremely large spatial data or performing complex spatial queries. Moreover, since GEUS fully supports object-relational models, its modeling power is outstanding compared to the other existing spatial database systems based on the relational models.

Key words
GIS, Spatial Database, Spatial Types, Spatial Operators, Spatial Access Methods, Spatial Query Optimization

1. INTRODUCTION

Remarkable growth of both database and GIS technology causes the need of spatial database that combines good points of both technologies. As current multimedia
technologies continue to improve, to serve multimedia information such as image, audio, video, and geographic information as well as alphanumeric information within database server shows a tendency to increase. On the other hand, GIS users accustomed to the features of existing databases expect that GIS should support many useful features of the typical databases. As a result, several architectural variants have been proposed to satisfy the users who want to handle spatial and aspatial data in a single database environment.

The architectures of GISs have evolved from the file based database model to the current object-relational database model since late 1970’s. Basically, GISs use two kinds of data: spatial and aspatial data.

In the first generation, spatial and aspatial data were separately stored in the files having uni-directional or bi-directional links. File system cannot support the concurrency control and recovery. Moreover, it cannot handle a large spatial data because of no existing optimized indexing schemes.

In the middle of 1980’s, a hybrid system was suggested. It stores aspatial data in the relational DBMS and spatial data in the file system (Becker, 1996). This configuration causes fatal issues such as integrity and security control of data managed by both independent systems. The problem in handling large spatial data also remains unchanged.

In the late 1980’s, to solve these problems, relational database model was proposed through the technical tie-up between relational DBMS and GIS vendors. Though this approach solved many problems of using file system for storing spatial data, such as concurrency control and recovery, it could not support efficient spatial data handling method because of its lack of spatial access methods and spatial data modeling ability. Hence GISs appended those capabilities at the outside of relational database by using database interfaces. However, this approach caused the degradation of system performance because of too many indirect access between spatial and aspatial attributes. It also needs a lot of disk space for connections.

In the middle of 1990’s, object-oriented database was proposed and applied to GISs to overcome the structural deficiencies of relational databases. Though this system had good modeling capability, it did not support structured query language (SQL). Thus users have felt inconveniences in querying spatial data.

This paper suggested object-relational database for storing spatial and aspatial data. By choosing object-relational database as the test bed for extension, we could take advantage of convenient and reliable features of both relational and object-oriented databases. Those are structured query language, concurrency control, and recovery mechanisms of relational database and the rich functionalities provided in object-relational database such as classes, inheritance, encapsulation, and methods (Stefanakis, 1996). Object-relational database is easy to model complex real worlds without any loss of information. Especially, inheritance and composition class features of object-relational database are very helpful to avoid unnecessarily duplicated data or interclass joins.

Section 2 gives the development methodology of the GEUS and its overall structure. Section 3 provides the spatial features of the GEUS including spatial types, operators, and access methods. Section 4 describes basic mechanisms of query
optimization for our spatial database in detail, and section 5 concludes this paper and
gives an overview of future work.

2. THE DEVELOPMENT OF GEUS

It is obvious that most users are likely to use a spatial database fully equipped with
the benefits of both typical DBMSs and GISs. This requirement caused many kinds of
affiliation between DBMS and GIS vendors. For example, Oracle made their relational
database support spatial data types and index structures through spatial data option, SDO,
while ESRI developed SDE in order to provide the management, access, and spatial
analysis functions. With SDE of ESRI, GIS data are stored within the Oracle. However,
this approach is accompanied with indirection burden, performance degradation, and user
inconveniences. From the first stage of development, therefore, we intended to solve
those problems. Thus GEUS tightly integrated the spatial functionalities with the object-
relational database engine. This approach makes GEUS conform to the conventional
database schemes and simultaneously provide spatial data types, operators, access
method, and query optimization.

2.1 Tight integration approach

Many attempts have been made to serve spatial data and their related
functionalities within the databases without any distinction between spatial and aspatial
data types. Representative two methods were proposed to realize those requirements. One
is to allow the users to add the spatial capabilities such as new data types and
functionalities including new access methods and query optimization algorithms. The
other is to make all the related database extensions to be fused into the database internals.
We call the former as a plug-in strategy, the latter as a tight integration strategy.

Plug-in approach contains complete misunderstanding in allowing the users to add
spatial access methods: it might be possible for the system to know the user-defined
spatial types. However, it is doubtful whether the query optimizer can recognize the
access method, and even if so, it is very difficult to evaluate the cost of the predicates in
association with the current database statistics at query optimization phase. Moreover,
transaction manager can not carry out recovery or concurrency control on index pages. So
we adopted a tight-integration approach which makes it possible to use spatial data types
and operators and to process query optimization in the same ways as in the conventional
non-spatial databases.

Key extensions to an object-relational database are as follows:First, parser was
changed to recognize extended spatial query language including the spatial data types,
spatial operators, and spatial access methods. Secondly, the internal type system was
extended to process the newly defined spatial types. Query optimizer was extended to
cope with the spatial predicates that include a spatial attribute and to determine an
optimal query(Ooi, 1989; Ooi, 1990). In addition to the query optimizer, we revised query
processor for processing a predicate involving spatial attributes and operators. Next, we equally situated spatial access method, which will decrease quite amount of time needed to access the spatial data, at the same position of B*-tree. R*-tree (Beckmann, 1990) was implemented as a spatial access method of our spatial database. Lastly, we integrated transaction manager (concurrency control and recovery manager) with the spatial access method, so as to allow automatic locking and recovery of index pages.

2.2 The structure of GEUS

Now let us look at the overall structure of GEUS in this section. Figure 1 shows the structure of GEUS. GEUS is logically divided into two parts: spatial query system and spatial storage system. Spatial query system is composed of query analyzer, query planner, and query processor. GEUS storage system uses spatial index scheme, R*-tree, for the fast retrieval of spatial objects.

Figure 1 GEUS Structure

Spatial query system of GEUS supports not only the conventional SQL but also spatial data types and operators within the SQL statements.
Query analyzer reads and analyzes SQL statements through interactive SQL or embedded SQL interfaces of GEUS. In general, most users who are familiar with the generic SQL statements can easily adapt themselves to our extended spatial query system. In section 3, spatial data types and operators will be discussed in detail. Query analyzer transforms the query statements into parse trees. Parse tree is going to be used for building query graph by query planner. In fact, query analyzer is composed of both syntactic analyzer and semantic analyzer. Syntactic analyzer builds skeleton parse tree and then semantic analyzer adorns the parse tree according to the definitions of database schema.

Query planner receives the parse tree from the query analyzer and then creates query plan in the form of query graph. Query graph informs query optimizer of the required statistical information of the interesting class and attributes such as the number of instances, the number of pages, or the fact whether the concerned attribute is indexed.

Query processor takes over the query graph from the query planner and executes that query according to the access plan as the query graph presents.

Query optimization process of GEUS is the one of outstanding strong points compared with any other plug-in systems unable to provide dynamically optimized plans. That’s because they only perform the cost estimation based upon the static selectivity of spatial operators kept on system catalogs. GEUS, however, is designed and implemented to find optimal query plan by dynamic evaluation based on the current statistics of database and selectivity of the spatial operators. In the case of hybrid query where spatial predicates with indexed spatial attributes are combined together with aspatial predicates with indexed aspatial attributes, query optimizer would compare cost value of each predicate and then decide which policy, spatial or aspatial index scan, will be applied before the other. Because GEUS always maintains the current statistics of the entire databases, there might be the case where aspatial index scan would be taken up first even if the selectivity of spatial operator is smaller than that of aspatial operator. Detailed considerations of query optimization processes will be discussed in section 4.

3. SPATIAL FUNCTIONALITIES

GEUS supports six basic spatial data types, nine topological operators, twelve geometric operators, and spatial path expressions. Those are tightly integrated with object-relational database engine. It is possible to define one or more spatial attribute(s) within a single class and also to add spatial indices, R*-tree, for each attributes. As GEUS automatically differentiates spatial access method from aspatial access method, users do not have to worry about syntax details with regard to their natures.

3.1 Spatial data types

Spatial data types are incorporated into internal type system of GEUS. They are primitive types and basically different from the system classes or user defined classes. As
a result, users can easily use spatial data types as normal aspatial types in their ad-hoc queries or applications.

We classified spatial data types into three categories: 0-dimension, 1-dimension, and 2-dimension. Point belongs to 0-dimensional category. Simpleline and polyline are the data types of 1-dimensional category. 2-dimensional category covers polygon, rectangle, and circle types. Table 1 shows the classification of spatial data types.

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Types</th>
<th>Spatial path expression components</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-dimension</td>
<td>point</td>
<td>x, y : double</td>
</tr>
<tr>
<td>1-dimension</td>
<td>simpleline</td>
<td>start, end : point</td>
</tr>
<tr>
<td></td>
<td>polyline</td>
<td>num : integer, start, end : point, mbr</td>
</tr>
<tr>
<td></td>
<td>polygon</td>
<td>num : integer, mbr : rectangle</td>
</tr>
<tr>
<td>2-dimension</td>
<td>rectangle</td>
<td>minpt, maxpt : point</td>
</tr>
<tr>
<td></td>
<td>circle</td>
<td>center : point, radius : double</td>
</tr>
</tbody>
</table>

Users can create classes as follows:

```plaintext
create class offices
 (name string, location point, address string, num_employee integer);
create class regional_headquarters (branches set of offices);
create class road (name string, road_num integer, path polyline, builder string);
create class city (name string, boundary polygon, population integer);
create class state
 (name string, boundary polygon, capital city, population integer);
```

As noted earlier, GEUS is an object-relational spatial database. So set functionality and composition class features are used to model real world.

### 3.2 Spatial operators

GEUS has representative two kinds of spatial operators. One is topological operator, and the other is geometric operator. We implemented topological operators which characterize spatial relationships between spatial objects (Bruns, 1996): contain, contained, cover, covered, crossover, disjoint, equal, overlap, and touch.

Egenhofer proposed the method for classifying topological binary relationships between area features. The classification is based on the definitions of his point-set topological spatial relations (Egenhofer, 1991; Egenhofer 1992; Clementini, 1993). Table 2 shows topological operators provided in GEUS.
Let us consider that a traveler wants to find out the cities crossed or touched by the road named ‘Kyungbu’, which is the longest highway running from ‘Seoul’ to ‘Pusan’ of Korea. In this case, we can use the SQL statement as follows:

```sql
select name
from city,
(select path from road where name = 'Kyungbu') as t(way)
where way crossover city.boundary or way touch city.boundary;
```

Note that t(way) is a temporarily derived table produced by the nested subquery. In addition, geometric operators were implemented. They can extract the node or the edge of a specific object, calculate the area of an object or the distance between two objects, or find out the nearest or furthest object within databases. Table 3 describes geometric operators and their descriptions.

When a user wants to know the area of a certain city or state, or the distance from the one city to the other, geometric operators can be used to meet such requirements. The following query statements would be applied to the various questions:

```sql
select nodes(boundary) from city;
select edge(path, 3) from road;
select name, area(boundary) from city;
```
select city.name, road.name
from city, road where distance(boundary, path) < 5;
select nearest(boundary, (select path from road where name = ‘Kyungbu’),1)
from city;

### TABLE 3

<table>
<thead>
<tr>
<th>Geometric operators</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>startpoint (Sobj)</td>
<td>start point of simpleline or polyline object</td>
</tr>
<tr>
<td>endpoint (Sobj)</td>
<td>end point of simpleline or polyline object</td>
</tr>
<tr>
<td>center (Sobj)</td>
<td>weighted center of an objects</td>
</tr>
<tr>
<td>nodes (Sobj)</td>
<td>all the nodes of simpleline, polyline, or polygon object</td>
</tr>
<tr>
<td>node (Sobj, n)</td>
<td>n-th node of simpleline, polyline, or polygon object</td>
</tr>
<tr>
<td>edges (Sobj)</td>
<td>all the edges of simpleline, polyline, or polygon object</td>
</tr>
<tr>
<td>edge(Sobj, n)</td>
<td>n-th edge of simpleline, polyline, or polygon object</td>
</tr>
<tr>
<td>nearest (Sobj1, Sobj2, n)</td>
<td>n nearest objects among Sobj1 from Sobj2</td>
</tr>
<tr>
<td>furthest (Sobj1, Sobj2, n)</td>
<td>n furthest objects among Sobj1 from Sobj2</td>
</tr>
<tr>
<td>maxdistance (Sobj1, Sobj2)</td>
<td>maximum distance between Sobj1 and Sobj2</td>
</tr>
<tr>
<td>distance (Sobj1, Sobj2)</td>
<td>distance between Sobj1 and Sobj2</td>
</tr>
<tr>
<td>length (Sobj)</td>
<td>length of an object</td>
</tr>
<tr>
<td>area (Sobj)</td>
<td>area of an object</td>
</tr>
<tr>
<td>direction (Sobj1, Sobj2)</td>
<td>angle between the straight line passing the two points center(Sobj1) and center(Sobj2) and X-axis</td>
</tr>
</tbody>
</table>

In addition to the above two types of operators, we devised a new operator, called spatial path expression available to retrieve the constituent values of spatial objects such as the x and y coordinates of the center point of a circle object or the start point of simpleline object. They have been described in table 1. Spatial path expression is represented by the dot operator used in the path expression of existing object-relational databases:

```sql
select boundary.start, boundary.mbr from city where name = ‘Seoul’;
select path.start.x, path.start.y from road where road_num = 101;
```

From previously mentioned above, newly defined spatial data types and operators not only conform to the existing SQL features of conventional databases but also make GEUS suffice all the requisites as a spatial database. Users who are accustomed to
handling databases with SQL statements can use the extended SQL systems of GEUS without other efforts.

### 3.3 R*-tree indexing as a spatial access method

As R*-tree index for spatial data types is incorporated in parallel with B+-tree index for aspatial data types, any distinctions or attentions for spatial index are not needed when creating indices on spatial attributes. If you would like to create spatial index on ‘boundary’ attribute of class ‘city’, you have only to write the following statement:

```sql
create index on city(boundary); /* R*-tree index */
```

Also the following statements are available in GEUS.

```sql
create index on offices(location); /* R*-tree index */
create index on offices(num_employee); /* B+-tree index */
create index on road(road_num); /* B+-tree index */
create index on city(name); /* B+-tree index */
```

Users can create one or more spatial indices on a class just as in the case of aspatial index, B+-tree.

So far, we have described spatial data types, operators, and access method of extended query system of GEUS. This discussion follows modification of query processing and optimizing mechanism. Especially, query optimization process plays an essential role to find an effective cost plan for processing a query by deciding an access path, a join ordering of multiple relations, a join method, and an evaluation order of query blocks(Aref, 1991). We will discuss about the basic spatial query optimization approaches and policies at the following section.

### 4. SPATIAL QUERY OPTIMIZATION

Although many external interface languages and data models have been proposed for spatial database, little attention has been paid to the optimization of hybrid queries resulted from the extended languages. In the earlier systems, evaluations of spatial predicates are always done before the conventional SQL predicates(Ooi, 1990).

The query optimization of GEUS is based on the cost model based approach of System R, which is an experimental database management system for the relational model of data and is developed by the members of the IBM San Jose Laboratory. For a complete description of the cost model based approach of System R, see (Selinger, 1979).

The query optimizer of GEUS examines both the predicates in the query and the access paths available on the classes referenced by the query, and formulates a cost prediction for each access plan, using cost formulas dependent on the type of predicates.
We denoted some key terms in conventional query optimization process before proceeding. The terms, NCARD, TCARD, ICARD, and NINDX are described as follows:

- NCARD(C) : the number of instances in class C
- TCARD(C) : the number of pages in class C
- ICARD(I) : the number of distinct keys in index I
- NINDX(I) : the number of pages in index I

These data are kept in a system catalog for every classes and indices. On the other hand, you need to know the fact that the predicates located in \textit{where}-clause is broken up into one or more conjunctive normal forms. Every conjunct is called a boolean factor. It is divided by spatial boolean factor and aspatial boolean factor once again. An index is said to match a boolean factor, if it has the form of “column operator value” and the column is the index key. Then optimizer assigns selectivity for each operator.

We intuitively inferred the selectivity for a spatial operator on the basis of the fraction of query region in the maximal coverage enclosing the entire spatial objects for the given class.

\[
\text{SELECTIVITY} = \frac{\text{Query Region}}{\text{Maximal Coverage}} \quad (1)
\]

Query optimizer would decide which formula is proper to be applied for cost evaluation under the current conditions. Query optimizer would first consider segment scan cost using the equation 2 where WCPU is an adjustable weighting factor between I/O and CPU.

\[
\text{COST} = \text{TCARD(C)} + \text{WCPU} \times \text{NCARD(C)} \quad (2)
\]

On the other hand, the cost of index scanning would be computed according to the following equation that is applied to estimate access cost for the predicates such as non-clustered boolean factors matching an index.

\[
\text{COST} = \text{SEL} \times (\text{NINDX(I)} + \text{TCARD(C)}) + \text{SEL} \times \text{WCPU} \times \text{NCARD(C)} \quad (3)
\]

Furthermore, we need to consider the spatial join involved with two or more classes. Basically, we support two kinds of join methods: nested-loop join and sort-merge join. Especially, in the case of spatial join, ordering of spatial value(MBR) is not clearly defined. So, GEUS supports only nested-loop join as a spatial join method. The cost formula for spatial nested-loop join is as follows:

\[
\text{COST-NESTED-LOOP} = \text{COST-OUTER(P}_1) + \text{N} \times \text{COST-INNER(P}_2) \quad (4)
\]

\text{COST-OUTER(P}_1) is the cost of scanning the outer class via path1, and N is the
cardinality of the outer class instances which satisfy the applicable predicates. \(N\) for 2-way join and n-way join are computed by equation (5) and (6), respectively. \(COST-INNER(P_2)\) is the cost of scanning the inner class via path\(_2\), applying all eligible predicates.

\[
N = (\text{the cardinalities of outer class}) \times (\text{the selectivity factors of applicable predicate}) \quad (5)
\]

\[
N = (\text{product of the cardinalities of all classes of the join so far}) \times (\text{product of the selectivity factors of all applicable predicates}) \quad (6)
\]

More complicated specifications regarding spatial joins are beyond the scope of this paper. Hence, access path selection for spatial joins would be treated when describing detailed query optimization methods of GEUS in the near future.

5. CONCLUSION

It is widely recognized that plug-in systems are unsatisfactory in providing the users with spatial features as in the conventional databases. In this paper, we presented the development of object-relational spatial database, GEUS. Since GEUS provides consistent mechanisms in dealing with spatial or aspatial data, users just need to know about what the augmented spatial data types and operators exactly mean and how to use them. As a result, we have been free from lots of inconveniences occurred in plug-in systems: constraints in defining spatial attributes and access methods within one class, unavoidable efficiency degradation, or overhead needed to interfacing separate GISs and database systems.

So far, we have discussed about how we implemented spatial database. However, much work has to be done constantly in order to improve system’s performance and availability. Also we have further research plans to develop parallel spatial database and spatiotemporal database.

REFERENCES


