# On Processing Nested Queries in Distributed Object-Oriented Database Systems

**Wang-Chien Lee**
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210-1277, USA

wlee@cis.ohio-state.edu, FAX: 614-292-2911

**Dik Lun Lee***
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong

dlee@cis.ohio-state.edu, FAX: 614-292-2911

## Abstract

*In this paper, we discuss nested query processing in a distributed object-oriented database system. We present three query processing strategies to exploit parallelism in a distributed environment. Then, we review three access methods designed for centralized systems and discuss how they can be applied to the distributed environment. Heuristics for selecting attributes for indexing and determining where to store the indexes are also presented. Finally, a replication strategy is proposed for the path dictionary method.*

## 1 Introduction

Different strategies supporting nested object query processing have been discussed in the literature. Several techniques, including indexing, signature file and path dictionary, have been proposed [2,3,4,6,7]. However, most of the work are for centralized systems; not much attention has been devoted to distributed environments. In this paper, we will discuss the research issues and propose query processing strategies and access mechanisms for nested queries in a distributed object-oriented database system (DOODS).

The rest of this paper is organized as follows. Section 2 is an overview of the object-oriented data model and the query processing strategies for centralized systems. Section 3 addresses system issues such as the distribution of objects and object identifiers. Then, we discuss the query processing strategies and indexing techniques for DOODSs in Section 4 and Section 5, respectively. Section 6 compares various query processing strategies. Finally, we conclude the paper with Section 7.

## 2 Query Processing for OODBSs

In object-oriented database systems, an entity is represented as an object, which consists of methods and attributes. Objects having the same set of attributes and methods are grouped into the same class. A class may consist of *simple attributes* (e.g., of domain `integer` or `string`) and *complex attributes* with user-defined classes as their domains. Since a class $C$ may have a complex attribute with domain $C'$, an *aggregation relationship* can be established between $C$ and $C'$. Using arrows connecting classes to represent aggregation relationship, a directed graph, called the *aggregation hierarchy*, may be built to show the nested structure of the classes. Figure 1 is an example of an aggregation hierarchy, which consists of four classes, `Person`, `Vehicle`, `Person_Name`, and `Company`. The class `Person` has three *simple attributes*, `SSN`, `Residence` and `Age`, and two *complex attributes*, `Owns` and `Name`. The domain classes of the attributes `Owns` and `Name` are `Vehicle` and `Person_Name`, respectively. The class `Vehicle` is defined by three simple attributes, `Id`, `Color`, and `Model`, and a complex attribute `Manufacturer`, which has `Company` as its domain. `Person_Name` and `Company` consists of two and three simple attributes respectively.

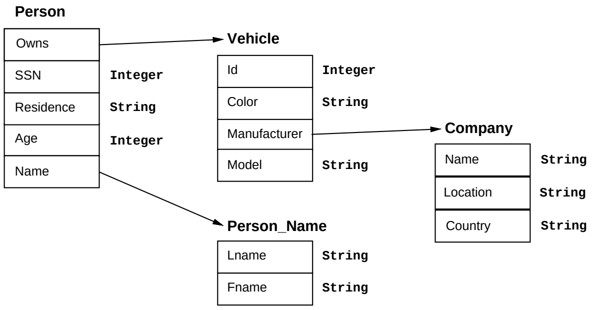


Figure 1: Aggregation hierarchy.

Every object in an OODBS is identified by an *object identifier* (OID). The OID of an object may be stored as attribute values of other objects. If an object $O$ is referenced as an attribute of object $O'$, $O$ is said to be *nested* in $O'$ and $O'$ is referred to as the *parent* object of $O$. Thus, child objects can be *forward referenced* from their parents through the OIDs stored in the parents. *Backward reference* from children to parents can be supported by explicitly creating a complex attribute in the child object referencing the parent (e.g., create a `owned-by` attribute in `Vehicle` with `Person` as its domain); alternatively, the system may implicitly maintain such inverse attributes. Objects are nested according to the aggregation hierarchy. Therefore, the aggregation hierarchy may be used as the schema of the object-oriented database.

Unlike relational databases, which use join operations to connect objects (tuples), OODBSs use OIDs

*The author is on leave from the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210.

embedded in complex attributes as the main mechanism in accessing nested objects. Navigation among objects through the OID links is called *traversal*. The navigational object access method is more effective than join operations since the OIDs provide direct access to the referenced objects while joins rely on value matchings. However, for object classes without aggregation relationships among them, join operations are necessary (e.g., to retrieve persons who live at the location of an automobile company). Since join operations have been thoroughly investigated for relational databases, we will focus on queries involving nested objects.

To facilitate our discussion, we call a query involving nested objects a *nested query*. We also define *target classes* as the classes from which objects are retrieved and *predicate classes* as the classes involved in the predicates of the query. Following Orion's query model [5], we assume that only OIDs from the target classes are returned to the user.

Nested queries may be classified based on the relative positions of the target and predicate classes on the aggregation hierarchy:

- TP: The target class is an ancestor class of the predicate classes, e.g., $Q_a$:"retrieve persons who own cars made by GM". In this query, `Person` is the target class and `Company` is a predicate class. The single-operand query discussed in [5] is a subset of TP queries.
- PT: The target class is a nested class of the predicate classes, e.g., $Q_b$:"retrieve manufacturers of the cars owned by persons at the age of 50".
- MX: The target class is an ancestor class of some predicate class and a nested class of some other predicate class, e.g., $Q_c$:"retrieve red cars owned by persons at the age of 50 and made by Ford".

The nested structure of an object suggests that answering a nested query requires traversal along the paths between the target class and the predicate classes. There may be more than one path in a nested query. For example, the query "to retrieve person objects with last name Smith who own a car made by GM" involves two paths, `Person.Name` and `Person.Vehicle.Company`. Since different paths have to be traversed separately, a nested query with multiple paths can be treated as a combination of several single path nested queries. Thus, we only discuss single path queries in this paper.

There are three basic approaches to evaluating a nested query: *top-down*, *bottom-up* and *mixed* evaluations. The top-down approach traverses the objects starting from an ancestor class to a nested class. Since the OID in a parent object leads directly to a child object, this approach is also called a *forward traversal* approach. On the other hand, the bottom-up method, also known as *backward traversal*, traverses up the aggregation hierarchy. A child object, in general, does not carry the OID of (or an inverse reference to) its parent object. Therefore, in order to identify the parent object(s) of an object, we have to compare the child object's OID against the corresponding complex attribute in the parent class. This is similar to a relational join when we have more than one child object to start with. Mixed evaluation is a combination of the top-down and bottom-up approaches, which is often used in query optimization as one of the alternative execution plans.

Both of the top-down and bottom-up approaches spend a significant part of the query processing cost on accessing intermediate objects connecting two objects. To alleviate this problem, many object accessing mechanisms, including indexes [1,2,3], signature files [4,7] and path dictionary [6], have been proposed. However, none of the above mechanisms have been applied to distributed environments. In this paper, we propose several accessing strategies and secondary organizations to support nested queries in distributed OODBSs.

## 3 Related Issues for DOODBSs

### 3.1 System Configuration

We assume that a distributed object-oriented database system consists of a collection of sites connected via a communication network. Each site has a host and a number of disks, which are treated as one logical disk. Each site is capable of data storage and management and may participate in query processing.

The database is distributed across the sites, each of which may issue a query and receive the result back. Each site maintains a copy of the system directory, which includes the global schema and system information such as the locations and distribution of the classes, objects and indexes on the sites. In addition, information useful for query optimization is kept in each site. For example, the work load at each host, speeds of the communication links, and statistics such as the cardinality of the classes, selectivity of the attributes, and access pattern of the queries, are maintained on each site.

### 3.2 OID Formats

In a distributed environment, an object residing on one site may reference objects on other sites through OIDs. Therefore, the object access mechanism must be able to rapidly access remote objects based on the OIDs. Two approaches, namely, *physical address* and *logical naming,* are typically employed for implementing OIDs. The physical address approach uses the object's physical address as the OID. In the logical naming approach, an OID has three parts: the site number, the class number and the object number. The object number is generated uniquely within a class on a site. Given an OID, the site number determines the storage site of the object. A table maintained by the object's storage site maps the OID's class number and object number into the physical address of the object.

The physical address approach is very efficient. However, in a distributed environment, where object migration could be very frequent, the physical address method is problematic since it requires all references to an object to be updated when the object migrates. For the logical naming approach, the problem with object migration may be resolved using forwarding addresses; periodic update to object references can be performed to eliminate long forward address chains. Also, this method allows us to determine the object's class from an OID without physically accessing the object from the disk. Therefore, we adopt the logical naming scheme in this paper.

### 3.3 Object Distribution

The distribution of the classes and objects among the sites is also an important factor for query processing. Typically, closely related classes and objects are put on the same site. In this paper, we consider two partitioning methods:

- Class Partitioning: different classes may be located on different sites.
- Object Partitioning: within a class, objects may be distributed on different sites.

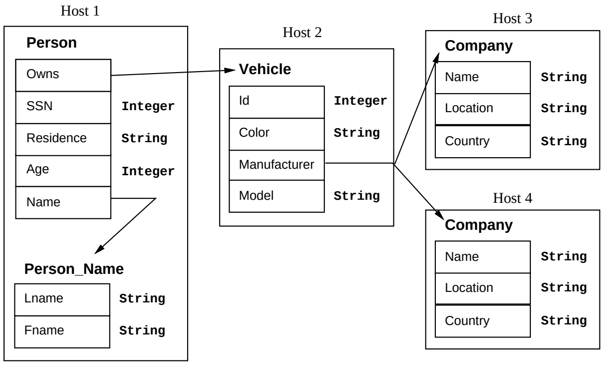


Figure 2: Database Partition.

Figure 2 is a partition of the database described in Fig. 1. In this example, the classes are distributed on different sites: `Person` and `Person_Name` reside on site 1; `Vehicle` resides on site 2; `Company` are stored on sites 3 and 4 based on the values of the `Country` attribute — objects with `Country` equal to 'USA' are stored on host 3; those without are stored on host 4.

In addition to class and object partitioning, a class may be vertically partitioned to expedite the retrieval of frequently accessed attributes. In addition, objects on the same site may be clustered to reduce the number of disk accesses for retrieving nested objects. In this paper, we won't consider vertical partitioning and clustering. We assume that an object resides entirely on one site since an object is a coherent entity and that objects of a class on a site are randomly stored in a file.

# 4 Query Processing for DOODBSs

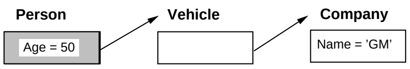


Figure 3: Query Graph.

In OODBSs, a query may be represented as a subgraph of the aggregation hierarchy containing only the classes specified in the query. Figure 3 shows the query graph for the query "retrieve person who is at the age of 50 and owns a car made by GM". There are three classes involved in the query. `Person` is both the target class and a predicate class. `Company` is a predicate class. The query graph may be decomposed into subgraphs for evaluation on different sites.

In this section, we discuss two factors which have profound impacts on nested query processing in a distributed environment.

## 4.1 Nested Object Traversal

In a distributed system, an object and its nested objects may be stored on different sites. As a result, access to nested objects might cross several sites, imposing extra communication cost and thus introducing a new variable for query optimization.

The forward and backward traversal approaches discussed in Section 2 can be applied to a distributed environment. Since the backward traversal approach is similar to join operations in relational database systems, most of the query processing techniques developed for distributed relational database systems are applicable to distributed OODBSs. For the forward traversal approach, the OIDs embedded in an object provide a direct reference to the child objects. The main difference is that the communication cost in a distributed system makes it especially important to have an efficient object traversal mechanism.

## 4.2 Parallel Processing

The concurrent execution of subplans on different sites may significantly speed up the evaluation of nested queries. There are basically three strategies for the concurrent execution of a query plan.

### 4.2.1 Parallel approach

In the parallel approach, the user site first divides the query into subqueries according to the classes involved. The subqueries, consisting of predicates, are sent to the storage sites of the classes in the subqueries. After executing the subqueries, the storage sites send the OIDs of the qualified objects and the OIDs and their nested objects to the user site for further processing. After receiving all of the OIDs, the user site performs the joins as in a centralized environment.

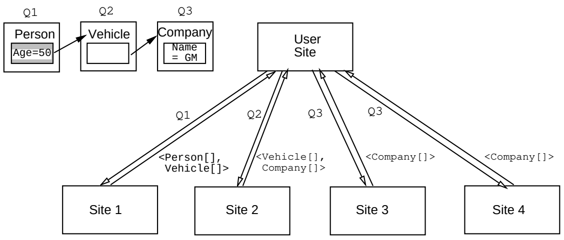


Figure 4: Parallel processing strategy.

Figure 4 illustrates the parallel strategy for the following query, $Q_d$: "retrieve persons who are at the age of 50 and own cars made by GM". In the figure, `Person[]`, `Vehicle[]` and `Company[]` represent sets of OIDs in the respective classes. After receiving the query, the user site decomposes the query into Q1, Q2 and Q3 and submits them to the sites corresponding to `Person`, `Vehicle` and `Company` for processing. Based on Q1, site 1 returns the OIDs of the persons at the age of 50 and the OIDs of their respective vehicles. Site 2 returns the OIDs of all vehicles and the OIDs of their respective manufacturers. Meanwhile, sites 3 and 4 return the OIDs of the companies with name "GM". At the user site, the sets of OIDs received from the storage sites are joined and the result is projected on `Person` to produce the final result.

The advantage of this approach is easy of implementation. Optimization techniques developed for relational joins may be used at the user site for the final join operations. Also, access to objects on different sites may proceed in parallel. However, the storage sites rely only on the local predicates to restrict the result set to be sent back to the user site; this leads to a large amount of information passed back to the user site for the final join operation. As a result, the user site may become a bottleneck because of the expensive join operations it has to perform.

### 4.2.2 Pipeline approach

The pipeline approach processes a query top-down in order to utilize the OIDs embedded in a parent object to access the child objects stored in another site. In general, this will restrict the size of the result set at the child site even when the query doesn't have any condition specified on the child class.

The query graph is decomposed by sending it to the storage site of the query graph's root class, $C_1$. In general, upon receiving a query graph, a site will retain the predicates specified on its local classes, record down the names of the local classes' child classes on the query graph, and send the rest of the graph to the storage sites of the child classes.

Processing starts from the root class's storage site, where the local predicates specified on the root class are evaluated. The embedded child OIDs needed by the next storage site are extracted from the qualified objects and passed onto the pipeline. The sites down the pipeline uses the OIDs received from the pipeline to directly retrieve the objects from their local classes for predicate evaluation, extract the OIDs needed by the next storage site from the qualified object and feed them on the pipeline. If the storage site holds a target class, it also passes the OIDs of the qualified target objects to the succeeding sites. At the end of the pipeline, the remaining target OIDs which satisfied the entire query are returned to the user site. This process is carried out in a pipeline. Once an object has passed the local predicate evaluation, the embedded OID can be extracted and passed to the next site for further processing. If transfering only one or one pair of OIDs each time is too expensive, several OIDs may be packed together to reduce the message transmit overhead.

Further optimization in the pipeline approach can be considered. For instance, a site may proceed with the local predicate evaluation without waiting for the OIDs to arrive from the pipeline; this is profitable if the local predicate is known to have a very high selectivity (i.e., very few objects qualify the predicate).

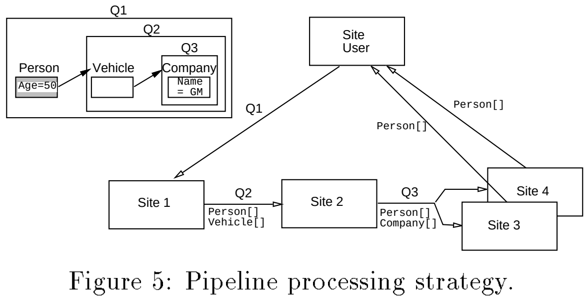


Figure 5: Pipeline processing strategy.

We use Figure 5 and $Q_d$ to illustrate the pipeline strategy. Since query decomposition takes a small amount of time, we assume that it is done before the pipeline starts to simplify our description. The pipeline starts at site 1, the storage site for `Person`. It sequentially retrieves the `Person` objects for evaluating the predicate "age = 50." The OIDs of the qualified `Person` objects and the OIDs stored in the complex attribute `Owns` are delivered to site 2. When site 2 receives a pair of `Person` and `Vehicle` OIDs, it uses the `Vehicle` OID to directly retrieve the `Vehicle` object and to project out the the `Company` OID stored in its `Manufacturer` attribute. The pair of `Person` and `Company` OIDs is then transferred to either site 3 or site 4, depending on the site identifier embedded in the `Company` OID. Finally, sites 3 and 4, using the `Company` OID, retrieve the `Company` object for evaluating the predicate "name = GM." If the car is indeed made by GM, then the `Person` OID is returned to the user site.

This approach is also very easy to implement. It balances the workload with cooperation among different sites in the system. The pipeline approach supports a high degree of parallelism. However, the path of traversal is predefined by the query plan, which might not be optimal.

### 4.2.3 Distributed approach

The distributed approach is less rigid than the parallel and pipeline approaches. It provides more options for executing a query and thus more opportunities for optimization. As in the parallel approach, the distributed approach divides a query into subqueries based on the classes involved. Then it dispatches the subqueries to the storage sites for further processing. Instead of sending the qualified objects back to the user site, the join operations proceed at the local storage sites. Unlike the pipeline approach, which has a pre-defined order of traversal, the distributed approach allows any pair of neighboring classes to be joined in either top-down or bottom-up fashion. The join operations proceed until the objects in the involved classes are all linked together. In other words, the join operations are executed in a distributed manner. This approach allows high degree of parallelism. In addition, optimization techniques such as dynamic programming may be used to estimate the cost for different join plans and to select the least expensive one. The site executing the final join operations will return the OIDs of the qualified target objects to the user site.

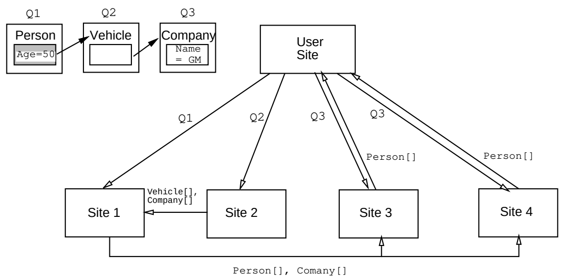


Figure 6: Distributed processing strategy.

Figure 6 demonstrates one way, which may not be the best way, to execute $Q_d$. Like the parallel approach, the user site dispatches subqueries Q1 to site 1, Q2 to site 2, and Q3 to both sites 3 and 4. Every site processes the subqueries they received. Since no predicates are specified on `Vehicle`, site 2 sends the entire set of `Vehicle` OIDs and the corresponding nested `Company` OIDs to site 1. Concurrently, site 1 retrieves the `Person` objects to check if their ages are 50. The qualified objects are joined with the OIDs received from site 2, and the OID pair for `Person` and `Company` resulted from the join are delivered to sites 3 and 4, depending on the site identifiers in the `Company` OIDs. Finally, sites 3 and 4 concurrently join the OID pairs received from site 1 to the `Company` objects with name GM. The `Person` OIDs are projected from the result and returned to the user site.

Since the parallel and pipeline approaches may be considered as special cases of the distributed approach, the distributed approach will produce the best execution plan, assuming that the cost of query optimization is not overwhelming.

## 5 Secondary Organizations for Query Processing

In centralized OODBSs, indexes, signature files and path dictionaries are the major secondary organizations for supporting nested queries. In a distributed environment, these secondary access mechanisms will play an even more important role since accesses across the network are expensive. In this section, we discuss several indexing and path dictionary mechanisms to support nested queries in distributed OODBSs:

- Indexes: Indexing techniques may be used to create implicit reverse links from an attribute to the ancestor objects. The ancestor classes are called *range classes* and the indexed attribute is called the *key attribute*. The index mechanisms reduce the amount of actual traversal through the network and the number of join operations needed. The nested index and the path index approaches are two examples of this technique. However, they only support backward traversal.
- Path dictionary: Instead of creating links in the backward direction, the path dictionary provides a bi-directional highway for traversing objects of different classes. Coupled with attribute indexes, the path dictionary can greatly reduce the amount of accesses to the database.

In the following, we use examples to illustrate the evaluation of nested queries with support of nested index, path index and path dictionary.

### 5.1 Nested Index

For a given path, the nested index maps the values of a nested attribute to the objects in the root class of the path. For example, a nested index may be created for the path `Person.Vehicle.Company` to map `Company.Name` to `Person`. Therefore, `Name` is the key attribute and `Person` is the range class for the index. The following is an example of the mapping:

| | |
|---|---|
| BMW: | `Person`[1] |
| GM: | `Person`[3], `Person`[5], `Person`[8] |
| Ford: | `Person`[2], `Person`[6] |

In the example, `Person`[$i$], where $i$ is an integer, denotes the OID of a `Person` object. A query such as $Q_a$ may be answered by sending the query to the resident sites of the index. The OIDs of the qualified `Person` objects can be returned to the user site after scanning the index.

For a query with predicates on the range class of the index (e.g., $Q_d$), traversal over the network is still needed. One way to answer the query is to send the query graph to the resident site of the index for index scanning. The `Person` OIDs obtained from the scan (i.e., persons who own 'GM' cars) will be delivered to the `Person` site for further predicate evaluation. Finally, the qualified objects (who are at the age of 50) are returned.

For queries with predicates which are not the range classes of any index, the nested index cannot prevent traversal but it helps in reducing the amount of communication. Consider the query, $Q_e$: "retrieve persons who own red cars made by GM". One way to answer this query is to use the index to find the OIDs of persons who own 'GM' cars and pass them to the `Person` site, where the `person` objects are retrieved and passed to the `Vehicle` site for predicate evaluation. Finally, the OIDs of the qualified persons are returned.

### 5.2 Path Index

Unlike the nested index, the path index maps the values of the nested attributes to lists of objects; each of the objects in the list belongs to a class in the given path. The linkages of the objects constitute the physical paths in the database. The range of a path index consists of all of the classes on the path. Thus, the scope of a path index is broadly extended. Assuming that a path index is created for the path `Person.Vehicle.Company`, mapping `Company.Name` to lists of objects corresponding to `Person`, `Vehicle` and `Company` classes, the following is an example of the mapping:

| | |
|---|---|
| BMW: | `Person`[1].`Vehicle`[5].`Company`[4] |
| GM: | `Person`[3].`Vehicle`[3].`Company`[2], |
| | `Person`[5].`Vehicle`[3].`Company`[2], |
| | `Person`[8].`Vehicle`[6].`Company`[2] |
| Ford: | `Person`[2].`Vehicle`[4].`Company`[7], |
| | `Person`[6].`Vehicle`[2].`Company`[7] |

The evaluation of $Q_a$ and $Q_d$ with the path index is similar to that of the nested index. For $Q_e$, however, the path index is more efficient, because the OIDs of the objects in the predicate classes may be obtained through index scanning. After scanning the index using "GM" as the key, the OIDs of the `Person` objects and the `Vehicle` objects corresponding to "GM" companies are sent to the `Vehicle` site for predicate evaluation. The OIDs of the `Person` objects with a red car are then returned from the `Vehicle` site to the user site.

### 5.3 Path Dictionary

The idea of path dictionary is to extract the path information (i.e., the physical linkage among objects) from the database into a secondary organization. This is done by storing in the path dictionary only the complex attributes of the objects. Since simple attribute values are not stored in the path dictionary, it is much faster to search the path dictionary than to traverse the objects in the database. Therefore, we can use the path dictionary to reduce the number of accesses to the database, and, in particular, to avoid accessing intermediate objects when we traverse from one class to another. In a distributed environment, since traversal among objects involves passing OIDs from one node to another for performing joins, the benefits of the path dictionary in a distributed environment is even more significant than in a centralized system.

In the following, we use examples to illustrate how nested queries are processed with the path dictionary. Assume that a path dictionary is created for the path

`Person.Vehicle.Company`. Figure 7 shows the information stored in the path dictionary:

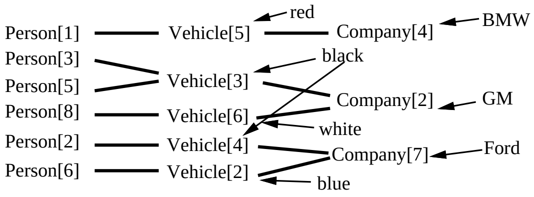


Figure 7: Path Dictionary Instances.

To implement the path information among the objects in a file, several organizations have been proposed, including *s*-expression, multi-links and path schemes [6,8]. The *s*-expression scheme represents the path information with subtree expressions. Take the path dictionary in Figure 7 as an example. The linkage information in the database can be represented in the following *s*-expressions:

`Company`[4](`Vehicle`[5](`Person`[1]))
`Company`[2](`Vehicle`[3](`Person`[3], `Person`[5]),
`Vehicle`[6](`Person`[8]))
`Company`[7](`Vehicle`[4](`Person`[2]), `Vehicle`[2](`Person`[6]))

On top of the path dictionary, indexes on arbitrary attributes can be built to map attribute values to the locations of the corresponding *s*-expressions in the path dictionary, thus avoiding a sequential search on the path dictionary. Since attribute indexes are very small compared to nested indexes and path indexes, a larger number of attributed indexes can be built for the same storage overhead.

To answer $Q_a$ with the path dictionary, the attribute index on `Company.Name` is scanned to locate the addresses of the *s*-expressions corresponding to the specified key value. The OIDs of the qualified `Person` objects can be obtained directly from the *s*-expressions. To answer $Q_d$, the `Company.Name` and `Person.Age` indexes are searched based on the predicates given in the query and the results are intersected. The OIDs of the qualified `Person` objects can be obtained from the *s*-expressions obtained from the intersected list. $Q_e$ is processed in the same way as $Q_d$. If the attribute indexes on `Person.Age` and `Vehicle.Color` are not available, then the path dictionary will be the same as the path index as far as $Q_d$ and $Q_e$ are concerned.

An important feature of the path dictionary is that it supports PT queries. Take $Q_b$ as an example and assume `Person.Age` is indexed. The attribute index for `Age` is scanned to locate the addresses of the *s*-expressions corresponding to the search key values. Then, the *s*-expressions are read from the path dictionary and the OIDs of the `Company` objects in the *s*-expressions are returned.

## 6 Issues with the Secondary Organizations

The nested index, path index and path dictionary are originally designed for the centralized systems. To expedite query processing in the distributed environment, there are several issues, such as storage location, partition, replication and update of the secondary organizations, to be considered. Possible change or extension of the organizations are under study.

### 6.1 Building Indexes and Path Dictionary

Two factors must be considered when we select an attribute for indexing:

- The attribute is frequently used in the predicates, since indexes built on frequently used attributes will benefit many queries.
- The mapping from the key attribute to the range classes is highly selective. That is, a key attribute value maps to only a few objects in the range classes. Since predicates on indexed attributes are most likely evaluated first, if the indexes produce very few qualified objects in the range classes, the amount of data transfer on the network will be greatly reduced.

Therefore, the choice of attributes for indexing should be based on two factors: the frequency of an attribute being used in queries, $F$, and selectivities of the attribute corresponding to the range classes, $S$. $F$ can be obtained by analyzing the cumulative statistics of the queries. If we assume a uniform distribution of the objects in the range class, $C_r$, on the index key values, $S$ may be estimated by $n/|C_r|$, where $n$ is the number of distinct values in the key attribute. For the path index, the selectivity of the index may be obtained as the average of the selectivities of the attribute corresponding to each of the range classes. Thus, $FS = F \cdot S$ may be used as a heuristics to choose attributes for indexing.

Another issue with building indexes is to decide the site for the installation of the indexes. To minimize the amount of data transfer, a nested index should be stored at the same site as its range class. For the path index, which may have multiple range classes, a site accommodating one of the classes on the path has to be chosen. Since the major concern with the index location is to reduce the cost of future predicate evaluation involving other classes, we want to choose a site with the following characteristics:

- the site accommodates many classes;
- those classes have many attributes frequently used in the predicates;
- the attributes of those classes have high selectivity corresponding to their resident classes.

The site with many classes means that, after searching the index, it is more likely for one of the classes to be involved in future predicate evaluation. If the index is installed at the site where many attributes are frequently used in predicates, the cost of data transfer among the sites may be reduced. So are the attributes with good selectivity. Therefore, we feel that the secondary organizations should be stored at the sites characterized above.

The heuristics we developed to choose attributes for indexing may be modified in order to be applicable to the selection of storage site for indexes. The selectivity $S_a$ of an attribute corresponding to its class $C$ is defined as the number of objects in class $C$ corresponding to a specific attribute value. Assuming a uniform distribution of the attribute values among its objects in $C$, $S_a = |C|/n$, where $n$ is the number of distinct values for the attribute. Therefore, the new heuristics for selecting the installation site of a path index is as follows:

$$\overline{FS} = \Sigma_{i=1}^{m} \frac{F_{a_i} \cdot S_{a_i}}{m},$$

where $a_i$ is an attribute of the classes stored on the site and $m$ is the total number of attributes.

Since the path dictionary are shared by attribute indexes, its storage cost is much lower than the path indexes. Therefore, it's affordable to attach many attribute indexes on the path dictionary. The choice of attributes indexing may use the same heuristics $FS$ we developed for indexing organizations. Meanwhile, the choice of storage sites for path dictionary may be decided by using $\overline{FS}$.

### 6.2 Partition of the Indexes and Path Dictionary

Data distribution introduces to the system advantages such as increased reliability and availability, data sharing, and better performance. However, distribution also increases the complexity in the system design and implementation. Since objects belonging to the same class might be stored on different sites (e.g., `Company` in our example), the organization of the indexes becomes complex. We address this problem in this section.

Since our discussion concluded that a nested index should be stored with its range class, let's consider the situation when the range class is distributed to more than one site. There are two basic approaches to building the index:

- Integrated index: Create an index mapping from the key attributes to the whole class and choose the site with the most objects to store the index.
- Distributed index: Treat the split classes as independent classes and create an index for each site which maps the key attributes onto the objects at the site.

The advantage of the integrated index is its simplicity: there is only one index to manage and maintain. However, when the range class needs further processing after index scanning, some of the object OIDs need to be delivered to the other sites for evaluation. On the other hand, the distributed index consists of multiple independent subindexes. The overall storage overhead is higher than that of the integrated index. However, during query processing, the indexes on different sites may be scanned concurrently. Also, no data transfer is required if further evaluation on the range class is needed.

Assume that, due to system expansion, the administrator of our example database decides to split the `Person` class on site 1a, including `Person`[1], `Person`[3] and `Person`[5], and site 1b, including `Person`[8], `Person`[2] and `Person`[6]. In the integrated approach, the index is not changed but rather stored either on site 1a or site 1b. In the distributed approach, the mapping of the nested index for the path is as follows.

| | | |
|---|---|---|
| Site 1a | BMW: | `Person`[1] |
| | GM: | `Person`[3], `Person`[5] |
| Site 1b | GM: | `Person`[8] |
| | Ford: | `Person`[2], `Person`[6] |

Assuming that the path index is also stored with class `Person`. The mapping of the distributed path index for the path is as follows.

| | | |
|---|---|---|
| Site 1a | BMW: | `Person`[1].`Vehicle`[5].`Company`[4] |
| | GM: | `Person`[3].`Vehicle`[3].`Company`[2], |
| | | `Person`[5].`Vehicle`[3].`Company`[2] |
| Site 1b | GM: | `Person`[8].`Vehicle`[6].`Company`[2] |
| | Ford: | `Person`[2].`Vehicle`[4].`Company`[7], |
| | | `Person`[6].`Vehicle`[2].`Company`[7] |

Similarly, the path dictionary may be distributed on several sites. The distributed path dictionary on each site will contain the path information about the objects reachable from the objects on the site. Figure 8 is an example of the distributed path dictionary. As shown in the figure, the attribute indexes built on top of the path dictionary should be separated too.

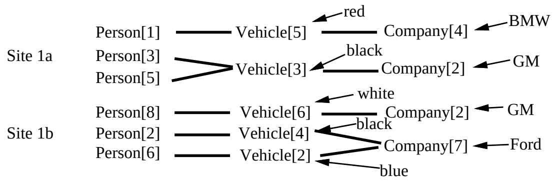


Figure 8: Distributed Path Dictionary Instances.

### 6.3 Replication of the Path Dictionary

If most of the frequently used attributes are indexed on top of the path dictionary, many queries may be answered on the site where the path dictionary is stored. Use Figure 8 as an example. To answer the query "retrieve persons who have red BMW cars," the query graph may be dispatched to site 1a and site 1b for processing. On both sites, the attribute indexes for `Color` of `Vehicle` and `Name` of `Company` are scanned to locate their corresponding sets of `Person` objects. The two sets of `Person` objects corresponding to "color = red" and "name = BMW" are intersected and returned to the user site, where the `Person` objects from site 1a and 1b are unioned and returned to the user.

The above query may be answered with a small number of page accesses and minimal data transfer. However, the storage sites of the path dictionary will be contended for answering queries, thus creating a bottleneck. One way to alleviate the problem is to replicate the path dictionary on several sites to split the work load. Another advantage of replication is that a query may be processed cooperatively by several hosts without communication among them. Let's reconsider the red BMW example. Assuming that all of the sites of `Person` and `Company` have the `Person.Vehicle.Company` path dictionary and `Color` and `Name` attributes on top of the path dictionary. Figure 9 illustrates the query processing process.

The user site may send subquery "retrieve person who own BMWs" to `Company` and send the subquery "retrieve person who own red cars" to `Person`. On these sites, the path dictionaries may be scanned concurrently and the OIDs of the qualified `Person` objects are returned to the user site. On the user site, the OIDs from `Person` sites are unioned and OIDs from `Company` sites are unioned. The resulting sets of OIDs are then intersected as the answer to the users.

Even though the path dictionary may be replicated for different sites, the information on different sites may

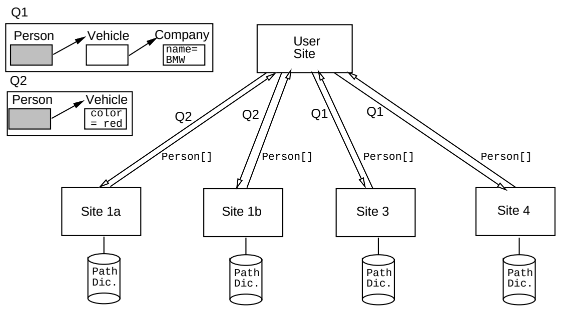


Figure 9: Query Processing with Replicated Path Dictionary.

be different. Use the instances in Figure 8 as example. Assuming that `Company`[4] and `Company`[7] are located on site 3 and `Company`[2] on site 4. The replicated path dictionaries for site 3 and site 4 are shown in Figure 10. Therefore, a path dictionary will contain the path information about objects reachable from the local objects.

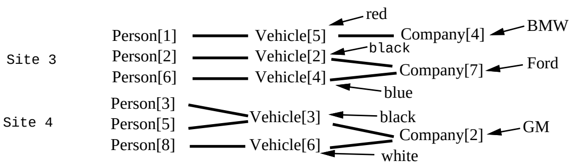


Figure 10: Path Dictionary for Company Sites.

Similarly, the nested index and path index may be replicated on several sites. However, not every site is suitable for replication in order to expedite query processing. The most reasonable site for the installation of an nested index is the site where its range class resides. As discussed, if the nested index is not located on the same site as its range class is, data transfer among sites is necessary for queries which need further predicate evaluation after index scanning. There may exist more than one range class for the path index. Therefore, the number of sites suitable for installing a path index are more than that of the nested index. However, the sites accommodating the classes nested in the class of the key attribute of a path index are not good candidates for storing the path index. That is because the backward reference nature of the indexing techniques doesn't use the nested classes as range classes. For example, a path index based on `Vehicle.Color` has `Person` and `Vehicle` as the range classes. `Company` cannot be part of the path index. Therefore, it is not suitable for the path index mapping from `Color` to `Person.Vehicle`. As a consequence, a query such as "retrieve persons who have red BMW" will need to traverse to the `Company` sites no matter where the path index for `Color` is located.

## 7 Conclusion

In this paper, we propose query processing strategies and support mechanisms for nested queries in distributed object-oriented database systems. Three query processing strategies, namely, parallel, pipeline and distributed approaches, were developed. Among these approaches, the distributed approach is expected to have the best performance, and the pipeline approach, with less communication overhead, is better than the parallel approach.

We discussed several approaches to using indexing organizations and path dictionary in nested query processing. We felt that the path dictionary is the most suitable secondary organization for the distributed environment. In addition to providing efficient support for distributed object retrieval, it also provides support for bi-directional traversals without causing much disk accesses and data transfer.

Another contribution of this paper is that we proposed heuristics for selecting attributes to build object access support mechanisms and for selecting sites to install these mechanisms. Among the three organizations we discussed, we found that the path dictionary is the best, because of its flexibility on selecting attributes and sites to index and low storage overhead when many indexes are to be created. Another noble characteristics of the path dictionary organization is its ease of replication for balancing the work load of the global system. We discussed an approach to replicate the path dictionary for split classes. A strategy for query processing with replicated path dictionary is also given.

## References


[1] E. Bertino, "An Indexing technique for object-oriented databases," *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, 1991, 160–170.

[2] E. Bertino, "Optimization of Queries using Nested Indices," *Proceedings of International Conference on Extending Database Technology*, Venice, Italy, March 1990, 44–59.

[3] E. Bertino & W. Kim, "Indexing techniques for queries on nested objects," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 2, June 1989, 196–214.

[4] Y. Ishikawa, H. Kitagawa & N. Ohbo, "Evaluation of Signature Files as Set Access Facilities in OODBs," *Proceedings of the 1993 SIGMOD Conference*, Washington, DC, June 1993, 247–256.

[5] W. Kim, "A Model of Queries for Object-Oriented Databases," *Proceedings of the IEEE International Conference on Very Large Data Bases*, Amsterdam, 1989, 423–432.

[6] D.L. Lee & W.-C. Lee, "Using Path Information for Query Processing in Object-Oriented Database Systems," *Proceedings of Conference on Information and Knowledge Management*, Washington, DC, Nov. 1994, 64–71.

[7] W.-C. Lee & D.L. Lee, "Signature File Methods for Indexing Object-Oriented Database Systems," *Proceedings of the 2nd International Computer Science Conference*, Hong Kong, Dec. 1992, 616–622.

[8] W.-C. Lee & D.L. Lee, "Path dictionary: A new approach to query processing in object-oriented databases," in preparation.