Performance Challenges in Object-Relational DBMSs

Muralidhar Subramanian, Vishu Krishnamurthy
Oracle Corporation
Redwood Shores
CA 94065

Abstract

Traditionally, relational database systems have excelled in handling relatively simple data, such as numbers, characters and dates. With the advent of the Internet, new programming languages such as Java, and new interchange languages such as XML, developers deal with complex data that need to be efficiently stored and managed. Extending Relational systems with Object capabilities has the potential to meet this challenge, but the handling of the complex data introduces new performance challenges that must be met by the resulting ORDBMSs. The principal performance challenges are found in the following areas: (i) handling collections of objects; (ii) handling domain specific data (e.g. Text, Spatial); (iii) handling Object References (e.g. Navigational Queries); (iv) handling Object Behavior; (v) mapping and handling objects on the client-side. In this paper, we will elaborate on the performance challenges in these various areas and discuss ways to address them.

1 Introduction

The tsunami of internet applications has brought newer and more complex kinds of data to hundreds of users around the globe. Object-relational technology has evolved relational database systems enabling them to store complex data: structured, unstructured or semi-structured. Once your objects live inside the database, you get to persist your object model, query without loss of encapsulation, and build robust and reusable database code.

Object-relational systems reduce the impedance mismatch between the application and the data stored in the database. Implementing the object abstraction layer on top of the relational infrastructure poses certain performance challenges.

Consider for example a typical e-commerce application that implements the purchase order system for a personal computer vendor. The data model for this system consists of:

- A purchase order object that contains a set of line items, a reference to a customer object, the shipping address, the order date and ship date.
- Each line-item contains a reference to the item, the quantity of order, any discount amount
- A customer object that contains customer name, customer address, customer credit information

Copyright 1999 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- An item object that captures the specification of a computer part (note that this may form an aggregation hierarchy of other parts/subparts), an image of the item, a text of the item description and warranty information, and the price of the item

An object-relational database management system (ORDBMS) supports the structure and complex relationships of the above data model such that one can query the database semantically and fetch objects from the database preserving their structure. For example, retrieving the purchase order for a given customer must return the order entry including all the line-items information. Or, for example, one should be able to query the purchase-order table to navigate the customer reference to obtain those orders that reference customers who live in California.

In addition to structure, a typical ORDBMS supports the ability to capture the behavior of objects as methods. For example, the purchase-order object may have a method that computes the total value of the purchase order. Capturing complex business logic as methods in the database enables applications to query and retrieve objects more meaningfully and efficiently. For example, the total value method enables the application to retrieve those orders whose value exceeds $1000. Contrast this to fetching the purchase order objects from the database to the client and performing the total value computation on the client-side to identify the purchase orders. Clearly, running data intensive application logic in the server has inherent performance advantages. [1] discusses the benefits of ORDBMSs and highlights the performance issues. [2, 3] are the benchmarks that exist server-side and client-side performance of ORDBMSs. In this paper, in the context of the above example we will highlight certain performance challenges that an object-relational system faces in support of the above modeling abstraction and offer some solutions.

2 Handling Collections of Object Data

Collections are essential to the object-relational type system to model multi-valued properties and relationships that are common in business applications. For example, the set of line-items in the purchase-order can be modeled using a collection type.

There are different ways of storing collections: as binary data or as rows in a table in an ORDBMS. Each method has its performance trade-offs. Storing collections by mapping its elements to rows of a table provides for better querying of the collection data as indexes may be created on these storage tables. For example, the line-items of purchase-order objects may be mapped to rows of a storage table by maintaining some referential keys between the purchase-order table and the line-items storage table. This storage model allows for efficient querying across line-items of all purchase-orders, for example, to retrieve those orders which contain a specific item.

However, if the collection has to be retrieved as an atomic value, then the ORDBMS has to construct the collections by aggregating the relevant rows of the storage table that constitute the collection value. Representing the collection as binary data, on the other hand, facilitates retrieval of the collection as an atomic unit since no construction needs to be done; however, querying of the collection is hampered since the binary data needs to be interpreted by the ORDBMS to read the collection elements. For example, the customer's list of phone numbers may be better represented as a collection stored as a binary value if the application always retrieves them as a single value and never queries its content.

It should also be noted that greater concurrency and finer level of logging is realized when the collection elements are mapped to rows of a storage table.

Representing large collections, such as time-series data, is another challenge faced by an ORDBMS. When an object containing such a collection is retrieved, it would be prohibitively expensive or nearly impossible to retrieve the collection data as part of the object's state. For these situations, an ORDBMS can provide the facility to return a locator to the collection instead of its value. Collection locators allow application to retrieve large collections without materializing the collections in memory. This allows for efficient transfer of such collections

across interfaces. In such situations, the ORDBMS must also support querying of these collections returned as locators to enable the application to retrieve a subset of the collection data.

3 Handling Domain Specific Data

Multimedia data types like images, maps, video clips, and audio clips were once rarely seen outside of specialty software. Today, many web-based applications need their database server to manage such data and provide the ability to query the content of such data and correlate it with other data in the database. Typically, database systems provide the way to store such content-rich data as binary streams in the database. However, efficient access and querying of such data pose a significant performance challenge.

Typical DBMSs support a few types of access methods (B+Trees, Hash Indexes) on some limited set of data types such as numbers, strings, etc. There is a growing need for indexing content-rich, domain-specific data, and also specialized indexing techniques. For simple data types such as integers and small strings, all aspects of indexing can be easily handled by the database system. This is not the case for documents, images, video clips and other complex data types that require content-based retrieval. Complex data types have application specific formats, indexing requirements, and selection predicates. For example, there are many different encodings (e.g., ODA, SGML, plain text) and information retrieval techniques (e.g., keyword, full-text boolean, similarity, probabilistic, and so on) for documents. Similarly, R-trees are an efficient method of indexing spatial data. No database server can be built with support for all possible kinds of complex data and their indexing requirements.

An ORDBMS should be an extensible server that enables the type definer to define new index types and operators in support of these complex data domains. The extensible framework of the ORDBMS should support the ability for the definer of the index type to specify its cost model so that the cost-based optimizer of the ORDBMS can evaluate the cost of the various access paths for a cross-domain query and select the optimal one.

In our sample application database, creating such domain indexes on the warranty text and the image of the item enables one to query the purchase orders to select those orders with item lists containing an item whose image matches a given input image and the warranty text contains a specific clause.

4 Handling Object References (Navigational Queries)

References in object-relational systems are the equivalents of primary key-foreign key relationships in relational systems. In addition to providing a more natural modelling of relationships between objects, references also allow users to specify more concise queries with path expressions (avoiding explicit specifications of multiple joins).

Navigational queries involve traversals through references. For example, to obtain the set of purchase orders of customers who live in California, the purchase order table is scanned, the target object of the customer reference is looked up - a process referred to as de-referencing the reference - and the predicate is applied on the address attribute of the customer object.

This, however, is not necessarily the most optimal solution. In many cases, the target objects corresponding to a set of references reside in the same table. Potentially better query plans can be generated if the optimizer is aware of this information.

A scoped reference is a reference whose target object resides in one well-known table - the scope table. De-references through scoped references can be converted into (outer) joins (on the object identifier columns) with the scope table(s). This allows the optimizer to pick alternate access paths for the query. For instance, it is possible that the predicates on the scope table are more selective than the predicates on the driving table - as is the case in our example above. Converting the operation into a join allows the optimizer to consider more optimal plans.

Scoped references offer storage benefits as well. An unscoped reference must contain an identifier for the table containing the target object of the reference. (This is not completely true. Systems that support global

database-wide indexes on object identifiers do not need table identifiers to be part of references.) A scoped reference does not need to, since the table containing the target object is well-known.

5 Handling Object Behavior

In an object-relational database, methods are the natural way to model object behavior. Regardless of what language the method is written in, method invocation is usually an expensive operation.

With the support for object types, methods run inside the server. Method performance in queries can be addressed by employing functional indexes and in-lining of method expressions

Functional indexes provide one way to improve method performance. A functional index, as the name suggests, is an index on the result of a function. During query analysis, the optimizer can identify functions that have indexes defined on them, and use the index access path for the function, rather than evaluating the function.

In many cases, object methods tend to be fairly simple. They are written as methods to provide encapsulated access to objects. For example, a *TimeForDelivery* method on the purchase-order object might simply compute the difference between the date the order was placed and the date it was delivered. Function in-lining is a useful mechanism to improve performance of such queries. In essence, with function in-lining, the body of the function is in-lined into the query, thus allowing the query optimizer to choose (potentially) more optimal plans. In addition, the method invocation overhead can also be avoided.

As we move from traditional business applications to newer domains, function logic tends to get more complicated. Optimizers in relational systems typically ignore the cost of evaluating functions. Object-relational databases should provide a way for users to register cost (and selectivity) information about functions, and then use this information to generate appropriate plans. Reordering predicates with expensive functions, and predicate migration are some strategies that can be employed in this regard.

6 Mapping and Handling Object data on the Client-side

Object data tend to be very structured and deeply nested. Applications fetch these objects from the database and navigate their relationships, in the process requiring other objects to be fetched from the database. Pure object-oriented database systems tend to be better at addressing this requirement since they support a memory persistence storage that inherently co-locates related objects on disk, and a page-based caching scheme that causes pages to be faulted in from disk to memory enabling several related objects to be fetched in one round-trip. Client-side management of objects pose a significant performance challenge for ORDBMSs whose storage is more relational.

In our purchase-order example, the application may wish to fetch the part hierarchy of a given item, make necessary changes, and persist the complex item in the database. Clearly, making separate roundtrips to the server to fetch every subpart of the item would be very inefficient.

Object-relational systems should provide a client-side object cache to aid in the memory management of the objects and provide a transparent persistency service. With the help of features like complex-object-retrieval (fetching and manipulation of a set of related objects as a single unit) into the object cache, clustering of related objects on disk, performance can be greatly improved.

With the help of an object cache, applications can “pin” an object reference to load the object from the server into the cache (if not already present in the cache) and “flush” to return and persist the modified object to the server. Storing the segment key or ROWID of the object as a hint in the object reference enables the ORDBMS to directly locate the object data on disk (without any index or table scans) in many cases. Updating the object in the client cache raises the issue of flushing the changes made by the client to the server. The object cache should provide an ability to flush all or selected updated objects to the server in one network roundtrip.

Acknowledgements

The authors wish to thank the following people for their valuable input and feedback: Jags Srinivasan, Nipun Agarwal, Ravi Kasamsetty, Ravi Murthy, Sandeepan Banerjee and Anil Nori.

References

[1] Object-relational DBMSs - Tracking the Next Great Wave, M. Stonebraker, P.Brown, Morgan Kaufmann, Second Edition, 1999.

[2] The Bucky Object Relation Benchmark (with M. Carey, J. Naughton, M. Asgarian, J. Gehrke, D. Shah), Proceedings of the 1997 SIGMOD Conference, Tucson, Arizon, May, 1997.

[3] The OO7 benchmark (with M. Carey and J. Naughton), Proceedings of the SIGMOD Conference, Washington, DC, May, 1993.