Query Processing in a Parallel Object-Relational Database System

Michael A. Olson   Wei Michael Hong   Michael Ubell   Michael Stonebraker
Informix Software, Inc.

Abstract

Object-relational database systems are now being deployed for real use by customers. Researchers and industry users have begun to explore the performance issues that these systems raise. In this paper, we examine some of those performance issues, and evaluate them for object-relational systems in general and for INFORMIX-Universal Server in particular. We describe object-relational query processing techniques and make some predictions about how they will evolve over the next several years.

1 Introduction

Since the middle 1970s, with the advent of the relational model and the appearance of working systems with which to experiment, academic and industry researchers have spent considerable time and energy inventing ways to make queries in relational databases run faster. Most relational database systems were sold to businesses, so most researchers concentrated on speeding up business query processing. The creation and use of indices, access path selection, cache structure and management, and the introduction of parallelism have all, over the years, attracted the attention of researchers. Innovations in these areas and others have produced a wide variety of fast, inexpensive, and powerful relational database systems.

Beginning roughly in 1985, with work on POSTGRES [Ston86] and Starburst [Haas90], researchers began to investigate a different kind of performance. Commercial relational systems were designed to execute queries quickly. These new research efforts were, fundamentally, designed to adapt quickly to changing business requirements, such as the need to model evolving business rules and manage new kinds of data. Systems of this kind, which have since been dubbed *object-relational*, preserved the relational model on which they had been built, but added support for developers to define new data types and operations. Object-relational systems have maintained backward compatibility with conventional relational engines. They can be used anywhere that relational engines can, because they provide the same operations and abstractions as relational engines. Object-relational engines use SQL3 [ANSI96], a superset of the query language supported by relational engines.

At the same time, object-relational systems borrowed heavily from work on object-oriented programming languages and systems.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Early object-relational systems emphasized extensibility over query processing performance. Since the beginning of the 1990s, however, and particularly with the introduction of Illustra to the commercial marketplace, object-relational systems have attracted the attention of the major database vendors and customers. Today, vendors are working hard to turn their relational offerings into object-relational products, and to make sure that those products are fast. In addition, industry and academic researchers are focusing their attention on query processing in object-relational systems.

In this paper, we draw on our industry and research experience to describe query processing strategies in object-relational systems, and to make some predictions about how such systems will evolve over the next several years. We summarize some of the major features of the INFORMIX-Universal Server, the first commercially- available parallel object-relational database system. We explain why object-relational systems are better than relational ones for high-performance query processing, and in particular how object-relational systems can take advantage of parallelism in interesting new ways.

2 Comparing Relational to Object-Relational

Because object-relational systems are still fairly new in the marketplace, discussion of them is often more vigorous than informed. In the sections that follow, we provide a brief comparison of relational and object-relational systems. This comparison provides a basis for evaluating both our claims and those of others on what object-relational systems can do.

A much more detailed comparison of the two architectures appears in [Ston96].

2.1 Object-Relational Query Processing

At base, object-relational systems use all the same query processing techniques as relational systems do. In particular, access path selection in object-relational systems works just like it does in relational systems. A cost-based optimizer considers a number of strategies for evaluating a query, and the least expensive plan is passed to the query execution engine. The executor supports the same scan and join strategies as do relational systems.

The most significant difference between relational and object-relational systems is that object-relational systems can always be extended with new code to store and operate on new data types. This extensibility is fundamental to object-relational data management, and permeates the entire engine. For example, an object-relational cost-based query optimizer must recognize the cost of operations provided by some extension, and must factor that cost into the total query cost.

Because object-relational systems are extensible, the number and variety of queries that they can execute is unbounded. Users may write queries over any data type, or combination of data types, known to the engine. As new data types are developed, they may be added to the system, and are available to users in the same way as those built into the system by the database system vendor.

A second important difference between object-relational and relational systems is that object-relational systems allow the definition of new primary and secondary storage structures. Developers may implement new ways of storing tables, and new index structures, to take advantage of special hardware, external software, or the properties of the data to be stored. A common example is the creation of spatial indexing structures to support fast searches on geographical or geometric data.

These new tables and indices give object-relational engines a potentially infinite number of ways of scanning user data. The query processing engine can choose an access path that uses a new index or table storage to execute a query. An object-relational query optimizer understands the performance characteristics of the new storage structures, and can use the same sort of cost metrics that relational systems do in order to produce a good query plan.

There are other ways in which object-relational engines improve on the query processing techniques afforded by strictly relational engines. Because they include all the components of a relational system, object-relational engines provide a strict superset of the query processing techniques available in relational engines.

2.2 Parallelism

All the major relational vendors provide support for *inter-query parallelism.* Inter-query parallelism allows multiple queries to execute in parallel, with some degree of resource sharing among them. For example, most vendors have a single shared buffer cache which all active queries use, and pages read into the cache by one query are available to any other query.

Some relational vendors also provide *intra-query parallelism.* Intra-query parallelism allows a single query to be split into multiple threads of execution, which can then run in parallel.

Because of their heritage, object-relational systems offer one or both of these varieties of parallelism. INFORMIX-Universal Server offers inter- and intra-query parallelism, as it is based on Informix's relational product, OnLine, and OnLine provides both kinds of parallelism.

Very few relational vendors offer *inter-function parallelism.* It allows a pipeline of functions in a query to be executed in parallel.

For example, consider an image processing pipeline, in which an image is clipped, scaled, and recolored to reduce the size of the palette it requires. A naive implementation of this process would be to clip the entire image, then scale the clipped version, then recolor the result. This strategy uses a lot of space, since at least two copies of the image must be in memory or on disk at each step. In addition, each function must wait for the one before it to complete before it can begin its work.

A better strategy is to use inter-function parallelism to change the way the pipeline is executed. Instead of operating on an entire image, each function in the pipeline operates on sets of scanlines. In this case, each function can be bound to a single thread of execution, and they can pass single scanlines among themselves. As a result, only a small amount of memory is required at each step, and the system can run the three operations in parallel.

Finally, no relational systems support *intra-function parallelism.* Intra-function parallelism allows a function over a single value to be broken into multiple threads of execution, so that each thread operates on part of the value at the same time.

Again, consider an image processing function. Scaling a large image up or down requires reading the entire image, but groups of scanlines can be scaled independently of one another. To scale an image down by a factor of two in both dimensions, for example, requires that each block of four by four pixels be replaced by a single pixel in the result. Computing the new pixel from the originals can proceed in parallel, since the result pixels are all independent of one another.

A well-designed object-relational system offers all these kinds of parallelism, because all are important to good performance. More importantly, the system must make all of them available to developers who write extensions. Inter-query parallelism is important because multiple users will want to use an image repository at the same time, just like multiple users want access to a payroll database managed by a relational engine. Intra-query parallelism is important to object-relational customers because object-relational databases can be very large, and only intra-query parallelism delivers good performance as the database grows. Intra- and inter-function parallelism are important because many popular extensions to object-relational systems support expensive functions on large data values.

3 Performance Issues in Parallel Object-Relational Systems

Object-relational engines allow developers, most of whom are not database server implementers, to extend the engine with new types and functions. Naturally, this power comes at some cost. Designers must consider the behavior and performance characteristics of each extension, and of combinations of these extensions, to guarantee that deployed systems will meet demands.

Query processing performance in object-relational systems has recently captured the attention of researchers. Work on characterizing performance has so far mostly used synthetic workloads, because object-relational systems have not been very widely deployed in commercial situations. As a result, the performance issues raised to date are interesting, but far from complete. Only with widespread production use of these systems will their real strengths and weaknesses become apparent. As a result, it is critical that object-relational engines continue to evolve, so that they can meet the demands of workloads that no one has yet foreseen.

With that caveat, [Dewi96] summarizes the most interesting performance questions confronting object-relational systems today. In the next several sections, we describe in detail how object-relational systems are able to overcome the problems cited in [Dewi96], and how they will change over the next several years as new problems arise.

3.1 Skew

Data skew is a well-known problem in relational systems. It arises when a system designer assumes that the distribution of data in a database system will follow one pattern, but the actual distribution follows a different pattern.

A common example of data skew is in the way that employees are assigned to departments in companies. Most technology companies, for example, employ many more engineers than accountants. As a result, the employee distribution is skewed toward engineering. If a database designer assumes that all departments will have the same number of employees, then query performance against this database may be bad.

Database systems that provide intra-query parallelism typically use data partitioning to provide distinct input streams for multiple threads of execution in a single query. This means that a single table may be spread across multiple disks, and that a single query can execute on each table fragment in parallel. The separate table fragments provide good performance because the system can read from multiple disks in parallel, which reduces aggregate I/O time, and because the threads can take advantage of multiple processors on the machine, if they are available.

Skew becomes a real problem in parallel systems when the distribution of data across disks is skewed with respect to queries. For example, if employees in our example are distributed across disks based on the department in which they work, then records for all engineers will be stored on one disk, and records for all accountants on another. Queries over all employees will not benefit very much from data partitioning, because the thread that examines engineering records will take much longer to complete than the thread that examines accounting records.

Relational database systems allow designers to partition data in three basic ways to handle this problem. Data may be partitioned in round-robin fashion, with new records being assigned to each disk in turn. It may be partitioned based on a hash value computed from the source key; this provides, effectively, random distribution, except that identical values are clustered on the same disk. Finally, data may be partitioned based on value ranges, so that employees who make between zero and twenty thousand dollars a year are assigned to one disk, with other salary ranges assigned to other disks.

[Dewi96] asserts that object-relational systems suffer more severely from skew than do relational systems. As an example, the paper draws on several novel features of object-relational systems, and describes how they can further skew distributions or lead to queries that do not match data distributions.

The INFORMIX-Universal Server uses data partitioning to provide intra-query parallelism and supports all the features that [Dewi96] uses in its example on skew. In order to understand the example, and why it is incorrect, it is important to understand the object-relational features that motivate it.

First, schema designers may create *row types*. Row types are records that may be stored and manipulated as atomic values in a column of a table. Thus a schema may include nested rows.

Second, a designer may use a *set* of values as a single value. Again, relational systems force designers to implement one-to-many foreign key relationships for a similar result. Object-relational systems make this easier to use by providing syntax and behavior that more closely matches the user's conceptual schema.

Using these features, [Dewi96] defines the following schema:

```
create row type emp_t (name varchar, salary money);
create table depts (name varchar, emps set(emp_t))
fragment by round robin in disk1, disk2, ...;
```

As in relational systems, if employees are not evenly distributed among departments, this schema is skewed. In fact, if we replace the set notation in this example with explicit join keys, and then fragment employees by department, we can produce an identical data distribution using a relational engine.

[Dewi96] claims that object-relational systems suffer because they cannot support fragmentation strategies that store sets on different disks from the rows that contain them. In fact, the behavior of any particular object-relational system is, in this case, a detail of its implementation, and not a fundamental feature of object-relational systems. Object-relational systems can support fragmentation strategies that distribute sets with, or separately from, the rows that contain them.

In addition, object-relational systems allow schema designers to use more than the simple hash, range, and round-robin fragmentation strategies offered by relational systems. Since an ORDBMS is extensible, designers may create arbitrary functions over their types that assign them to buckets or value ranges based on knowledge of the types' structures or the likely distribution of real data. Schema designers may do hash- or range-based partitioning on the result of a function, not just on an atomic value. The same statement applies to values in sets, or in any other SQL3 collection type.

The important point is that schema designers can use the declarative syntax with which they are comfortable to control the partitioning of both tables and collections. In general, that will be sufficient to create a schema that performs well in practice. In some cases, a designer may have special knowledge of the data domains or distribution that a database will store. In those cases, the schema designer can easily add new classification or hashing functions to the system, and can use those functions to control data placement.

Furthermore, users automatically get the performance improvements of a parallel system over new data types. The engine understands how to provide both query and function parallelism. Developers who write extensions need not do anything special to take advantage of it. Users who query the system via SQL need not do anything special to take advantage of it. The support is built in, and invoked automatically.

Note that skew, and in particular the kind of unequal data distribution produced by the declarations above, is not new to ORDBMSs. These problems have been noticed and studied carefully in parallel relational systems for years. An ORDBMS provides much more control to a schema designer to handle skew and do intelligent data distribution than does a conventional RDBMS.

3.2 Bad Schema Design

To alleviate the problems in the schema declared above, [Dewi96] proposes creating separate tables for employees and departments, and partitioning both of them in round-robin fashion across disks in the database.

Given that schema, [Dewi96] correctly demonstrates that some queries will perform badly. In particular, if employee records are not clustered with the department records that contain them, then a query that computes the average salary of employees by department turns into effectively random I/O across all disks, and one message per record among the threads participating in the query.

Naturally, performance is terrible.

However, this problem is not due to any object-relational features. Rather, it is an example in which poor schema design leads to very bad query performance. Any physical organization of data on disk must consider the most common user queries, and data should be clustered to optimize those queries. This requirement has long been acknowledged for relational systems, and applies just as strongly to object-relational ones.

The basic problem with [Dewi96] is that it partitions data to support a particular workload, and then evaluates the schema using another workload entirely. Any system can provide at most one clustering strategy for data without copies. An object-relational system outperforms a relational one, if the two systems have identical data partitioning and query plans, because the object-relational system allows designers to choose among several models, such as explicit joins versus references, for performance or representational reasons.

3.3 Large Values

Finally, [Dewi96] explores performance of queries that operate over large data values by using three examples. Those examples provide a good demonstration of the superiority of object-relational systems to conventional ones, so we will preserve them in this discussion.

3.3.1 Time Series Data

The first example is time series data, such as information from a stock ticker. At regular or irregular intervals, new information arrives on the ticker, and should be included in the appropriate large time series. The SQL DDL

```
create table stocks {name varchar, prices timeseries(money));
```

creates a table that captures all the ticker traffic for a given stock name in a single record. As new price information arrives over the ticker, it is added to the appropriate timeseries value in the table.

Given this schema, for example, a user could write a query that computes the average closing price of a particular stock. [Dewi96] makes the point that if values in a timeseries are not partitioned across disks, then this query will get no benefit from parallelism.

That claim is correct, but is not an indictment of object-relational systems. Just as sets may be partitioned independently of the records that contain them, so may other constructed values. A timeseries, an array, a set, and a collection are all examples of constructed values, and users may want to partition the values stored in any of these. By partitioning timeseries values across disks, schema developers can take advantage of intra-function parallelism. Relational systems, which do not even support the new collection type, cannot offer the same functionality or performance.

3.3.2 Video

The next example in [Dewi96] uses video to argue that object-relational systems will not scale. If videos must be stored inside the database system, then playback speeds are likely to be compromised (since no object-relational system is designed to provide constant-bit-rate services), and operations on single videos will be slow, since videos cannot be partitioned.

The first claim is certainly correct; a video management strategy that stores videos in binary large objects should not be used for playback. In fact, any video management system that stores videos inside the database is likely to collapse under the weight of the data it holds. The files are huge. It takes only a few tens of minutes of compressed video to fill a two-gigabyte disk completely.

Object-relational systems allow the database to refer to values stored externally, on video servers or other special purpose hardware. The ORDBMS can fetch the contents from the repository when it must operate on them. The database engine stays out of the data path for playback. Naturally, there are issues of referential integrity and access control to consider, but the important point is that an ORDBMS is flexible enough to support both local and remote management of large data values.

3.3.3 Images

The third, and final, example type is image data. In this case, [Dewi96] claims that queries operating over images will run slowly, because the system has no way to parallelize expensive operations on single images.

In fact, an ORDBMS offers a number of ways to exploit parallelism. INFORMIX-Universal Server uses the strategy from Volcano, discussed in [Grae90], to represent in the query plan the points at which parallelism is possible. A type or function designer only needs to create a special class of function, called an *iterator*, in order to allow intra-operator parallelism. The INFORMIX-Universal Server considers available system resources and global optimization in order to decide where, and how, to exploit parallelism.

The best way to parallelize a sequence of operations over large data values is to break the value into a set of smaller values, and to have the pipeline operate over the members. For example, rather than writing functions that take and return a single image, a programmer can write functions that take and return a set of pixel rows. In that case, a sequence of operations, like scaling a clipped, resampled image, can execute in parallel, and only a small amount of data must be in memory at any time.

Once again, an ORDBMS provides designers with a variety of options to control the performance of their queries. The engine itself imposes no particular restrictions on how values are represented, and is able to exploit inter-query, intra-query, inter-function, and intra-function parallelism.

4 Conclusions

Object-relational systems are poised to replace relational systems in large-scale deployment over the next several years. As a result, researchers have begun to investigate the performance characteristics and design trade-offs in such systems.

Our experience in building, shipping, and supporting industrial-strength ORDBMSs convinces us that they are superior to their relational predecessors. This superiority is based on no single feature, but rather on the extreme flexibility of object-relational engines. They have been designed to support rapid innovation and maintain good performance on relational workloads. As a result, it is simple to experiment with ways of improving performance whenever new bottlenecks are discovered.

Object-relational systems give developers the tools that they need to build systems that perform well. A type designer, for example, can choose to create types and functions that support intra-function parallelism by using collection types, and building higher-level structures like images on top of lower-level structures like scanlines.

Similarly, schema designers have much better control over physical layout and logical data model than they do in relational systems. Because collection values can be partitioned independently of rows that contain them, the designer can lay out data to balance load across disks and to provide excellent query response for the most commonly-asked questions.

We believe that the currently-shipping object-relational systems address the performance problems raised in the research community now. In addition, because of the attention that object-relational systems must pay to extensibility, they will continue to perform well as experience grows and our understanding of workloads matures. This is because the constituent parts of a relational engine, like the query parser, planner, executor, and storage management, are well-separated and independent in an object-relational system. As a result, changes to the internals of any single component are less likely to affect the others, making rapid modifications easier.

References

[ANSI96] ANSI X3H2 Committee. *Database Language SQL - Part 2: SQL/Foundation*, Part 2. Committee Draft, July 1996.

[Dewi96] Dewitt, David. Object-Relational and Parallel: Like Oil and Water? *Proceedings of Object-Relational Summit*, Miller Freeman, San Francisco, 1996.

[Grae90] Graefe, G. Encapsulation of parallelism in the Volcano query processing system. *Proceedings of ACM SIGMOD Conference*, ACM, New York, 1990.

[Haas90] Haas, L., *et al.* Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering* 2(1), March 1990.

[Ston86] Stonebraker, M., and Rowe, L. The Design of POSTGRES. *Proceedings of the 1986 ACM-SIGMOD Conference*, ACM, New York, 1986.

[Ston96] Stonebraker, M., and Moore, D. *Object-Relational Databases: The Next Great Wave*, Morgan Kaufman, San Francisco, 1996.