Java in Next-Generation Database Systems

Praveen Seshadri
Computer Science Department, Cornell University
praveen@cs.cornell.edu
http://www.cs.cornell.edu/home/praveen/

Abstract

This position paper describes the impact of Java on the next generation of database systems. These observations are based on our ongoing research in building the PREDATOR object-relational database system at Cornell. We believe that Java will play a critical role in two complementary components of the DBMS: (a) user-defined extensions of the database server, (b) migration of query processing between server and client. We are beginning to explore these topics in the context of the *Jaguar* project.

1. Introduction

Many interesting web applications require the involvement of HTTP servers and database servers. While most of the current research is on enhancing the performance of HTTP servers, we focus instead on the challenge of adapting database servers to operate efficiently in a Java-enabled internet environment.

Current relational and object-relational database systems are typically built with a query-shipping client-server architecture. Queries are shipped from the client to the server, where they are processed, and the answers are sent back to the client for display. Over the last decade, database servers have also developed extensibility features --- new data types and new functions can be added to the system in a modular fashion[7,8]. These two trends in database software architecture need to react to recent advances in web technology. The focus of this position paper is on the deployment of extensible client-server database technology in a Web-based user environment. Specifically, we make the following observations:

1. Web browsers are universal, making them the ideal client interface for all interactive applications, including database applications. This brings a much wider audience in direct contact with database systems.
2. Most browsers support the Java Virtual Machine[12], which executes Java byte codes. Consequently, Java has become a desirable implementation language for clients.
3. Because of the large number of users with diverse needs, the importance of extensible database systems has increased. The ease of extension also becomes an important issue, because most of the users do not have access to the database server, the machine on which it runs, or the software environment and language in which the server is written.
4. The users of the database are located at varying distances (measured physically and/or by network communication time) from the server. Consequently, simple query-shipping mechanisms are not necessarily optimal. Instead, the boundaries between client, server and network need to be blurred so as to derive efficient query processing performance.

This Web-based environment leads to new challenges for next-generation database applications. Extensibility mechanisms have been studied in the context of database support for complex data types[8]. Client-server query processing has been studied in the context of homogeneous distributed systems[9]. What is needed is an integrated approach, knowing that the applications run over the Web with Java available as a portable programming environment.

2. Background

This section contains a brief overview of database technology related to this position paper.

Database Deployment in a Web Environment

The architectures of web-based database applications fall into two broad categories: Two-Tier and Three-Tier architectures. In both categories, a database server runs on a machine accessible via the Internet, and users interact with web browsers on their local machines. In a Two-Tier architecture, the database client is a Java applet that runs within a web browser. It directly connects to the database server, sends requests to the server, and displays the results to the user. Figure 1 shows a schematic of such an architecture. In a Three-Tier architecture, the client is devoted to providing a suitable HTML-based user-interface, while the "application logic" that translates user inputs into database interactions is implemented in a "middleware" layer that binds the HTTP server and the database server. The issues raised by this paper apply to applications built with either architecture. However, for the sake of simplicity, we will assume a Two-Tier architecture.

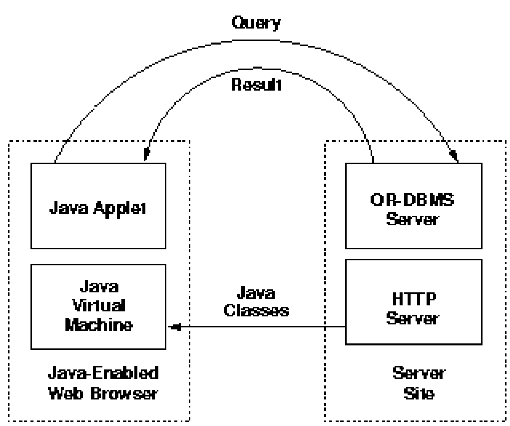

Figure 1: Sample Two-Tier Architecture

All the large relational database vendors sell products that facilitate Web/DB connectivity using a multi-tier architecture. The JDBC protocol [11] establishes an important step towards Java-based database clients; it specifies a standard Java interface using which an application can interact with any database system, independent of vendor-specific details. However, the JDBC protocol essentially ignores the issue of database extensibility.

Work on Client-Server Query Processing

Distributed database systems and client-server systems use a variety of query processing techniques. These techniques have traditionally been categorized as either "query-shipping" or "data-shipping"[9]. Query-shipping is common in relational and object-relational databases, where clients send entire queries to the server and wait for the results to be sent back. Data-shipping is common in object-oriented databases, where the server is simply a data repository and all the query processing is done at the clients. Another proposal is a "hybrid-shipping" strategy, which combines elements of data-shipping and query-shipping. Researchers in the DIMSUM project [9] explored hybrid-shipping strategies primarily with the motivation of exploiting cached data at the client. In web-based applications, cached data is likely to arise in the middleware layer of Three-Tier applications. The database query optimizer needs to be augmented to consider hybrid-shipping query plans in addition to traditional plans. We believe this to be a crucial feature of Web-based query processing systems.

Work on Database Extensibility

Since the early 1980s, database servers have been built to be extensible; that is, to allow new application-specific functionality to be incorporated. Most commercial extensible DBMSs and large research prototypes support user-defined functions (UDFs) that can be added to the server and accessed within SQL queries. The motivation for server-side extensibility is efficiency; a user-defined predicate might greatly reduce query execution time if applied at the early stages of a query evaluation plan at the server. Given the focus on efficiency, most research on UDFs has investigated the interaction between database query optimization and UDFs. Specifically, cost-based query optimization algorithms have been developed to ``place'' UDFs within query plans[3,4].

Traditionally, it is assumed that most database extensions are written by authorized and experienced ``DB Developers'', and not by naïve users. Extensible database systems usually provide three options to those customers who prefer to write UDFs themselves:

(a) incorporate UDFs directly into the server (and incur the substantial risks that this approach entails),

(b) run UDFs in a separate process at the server, with some simple operating system security guarantees, or

(c) run UDFs on the client-side in a client environment that mimics the server environment.

The PREDATOR database system

PREDATOR is an object-relational database system developed at Cornell. It is a highly functional system, capable of supporting a large subset of SQL-92, multiple concurrent transactions, and database applications involving gigabytes of data. It exploits the storage management functionality of the Shore Storage Manager software library[1]. PREDATOR has primarily been used for research into efficient techniques for the support of complex data types [2]. It has a client-server architecture; clients are expected to acquire queries, ship them to the server, and display the returned results. A Java client library (JCL) has been written that provides a superset of the functionality of JDBC[11]. The JCL can be used to build application clients either as middleware programs or as applets that run within a Web browser. This position paper arises out of our observations in building database applications over the Web.

3. User-Defined Extensions

Consider a database application, where there are many users, each desiring to extend the system with his/her own Java UDFs. Several design issues arise:

- *Security*: Are the UDFs trusted code? What security mechanisms are needed?
- *Efficiency*: How does the security mechanism affect the performance of queries?
- *Usability*: How portable are the UDFs? Do the security mechanisms adversely affect the ease of extensibility by users?
- *Query Processing*: What changes to the query processing engine are required to use portable UDFs?

Ideally, it should be possible (and straightforward) for a naive user in a web environment to create a UDF, to test it at the client site, to register it with the server, and to have the server authenticate and execute the UDF code. We identify four broad areas of research on this topic:

1. Different mechanisms (include Java extensions) exist to support database extensibility. What are the tradeoffs between these alternatives with respect to efficiency?
2. How do Java extensions impact the security and reliability of the server?
3. The obvious attraction of Java is its portability and universal availability. However, internal database interfaces still need to be designed appropriately to permit transparent application of Java extensions at the client or at the server. What are the suitable interfaces that accomplish this location transparency?
4. What changes are needed to the database query optimizer to consider and cost the new query processing alternatives that involve migration between server and client.

We have done initial work in exploring the first couple of issues[14]. While the obvious security mechanisms address the safety of memory accesses and control transfers, the memory, CPU, and I/O resource usage of individual extensions are difficult to monitor or police. Thus, simple denial-of-service attacks (or inadvertent resource over-consumption) are possible. This is one of the issues currently being addressed in the J-Kernel project[13].

The third topic may initially appear to be a mundane implementation detail. However, we have realized that the UDF invocation mechanism (specifically, the parameter passing mechanism to a UDF) requires non-trivial design. Our goal is to allow a user to specify a UDF using Java code developed at the client, and to run the UDF within an SQL query at the server. Actually, there are three broad execution choices for such a query:

1. The query can run totally at the server. In this case, the UDF must be invoked from the server.
2. The query can be run totally at the server, except for the UDFs, which are applied as a "post-processing" filter at the client. This case occurs with UDFs that are local to the client and have not been migrated to the server (for instance, because they are currently being developed and debugged).
3. The query is run totally at the server, except for the UDFs, which are evaluated at the client upon request by the server. In other words, during query execution, the server may ask the client to

execute the UDF, and then use the result of that execution to continue processing.

It is clear from these choices that the very same UDF code needs to run at both client and server at different stages of the develop-test-use cycle. There are two components that affect a UDF execution; the code being executed and the parameters passed to it. Since the code is a Java bytecode program, it should behave similarly at client and server. Similarly, data parameters need to be converted into a common form that can be interpreted at both ends. This requires careful interface design at both the client and the server.

How should a query optimizer choose between these different alternatives? It is only relatively recently that relational query optimizers developed techniques to appropriately place UDFs within query plans based on their cost. In the case of client-side UDFs, it may be appropriate to delay their execution in the query plan, because of the cost of crossing the client-server boundary. Even when the UDFs can be run at the server, it may be preferable to delay their execution and run them at the client, because that strategy does not need to tax the server's resources by running the UDFs which can be run on the client.

4. Portable Query Processing

So far, we have been motivated by the desire to migrate UDF code from the clients to the server. In order to accomplish this target, the UDFs are implemented in Java, with its portability and security benefits. We now consider the dual motivation: is it desirable to migrate traditional server-side query processing to the client, even in the absence of UDFs? There are three general situations in which this is desirable:

1. The client has idle resources (like memory and CPU) which are in great demand at the server. Consequently, some part of the query computation (and the data on which to perform it) could be shipped from the server to the client.

2. The network is slow and the transfer of bytes over the network is a critical factor in the overall response time observed by the user. In this situation, some intermediate stage of query execution may form a smaller data set than the final answer. Consequently, it may be cheaper to ship this intermediate data set to the client, along with the remaining query computation.

3. The client may have cached some of the data needed to execute the query, or the result of earlier queries that are sub-components of the current query.

We need to design and implement practical solutions that exploit the optimization opportunities that exist in all these situations. The second situation requires some explanation. The most common queries involve joins of data tables. The size of the result of a join may be greater than the sum of the sizes of its input tables (we term this an "expanding join"). In such cases, it may be preferable to move the join computation to the client. In fact, a single query may involve the combination of several of these situations. There may be UDFs, expanding joins, and available client-side resources.

We address two broad topics on which research should focus: (a) the design and implementation of the infrastructure to allow hybrid-shipping of queries, (b) the query optimization issues that arise.

We believe that hybrid-shipping in a web-based database environment requires a major change in the software architecture of database servers. This belief is based on the following argument:

(a) Java is the obvious language choice for portable query plans,

(b) the lower levels of database servers will continue to be implemented in a language like C or C++

(c) the query optimizer and query processing engine will have to be significantly altered in order to support this combination.

Let us consider this argument in more detail. At first glance, it appears that hybrid-shipping merely requires that query plans (or rather, components of query plans) be represented in some language and machine independent form. Both client and server need to have an "interpreter" for the query plan – it is desirable that the same interpreter codebase be used at both client and server. The interpreter includes code to perform joins, aggregations, etc. Clearly, the client-side interpreter would be implemented in Java for widespread usability. At At the server, an interpreter written in Java is preferable to supporting a parallel codebase that implements the interpreter in another language.

Does this imply that the entire database server is written in Java? There are some current research and commercial projects attempting to build such "all-Java" systems (for example, [5]). However, we believe it is undesirable (for performance reasons and practical reasons of inertia) to implement many of the lower levels of the server in anything other than a language like C or C++. Instead, we propose to modify the query execution engine, but not the rest of the database system. Figure 2 sketches an outline of how this strategy would work. The query optimizer generates a query plan, which is then interpreted using a collection of Java classes. On the server, these classes interact with other "native" support classes that provide access to the lower-level functionality at the server (storage management, etc.). The very same query plan and plan evaluation code can also run at the client. However, there are separate support classes that provide the same abstractions as the support classes at the server. Is such a software architecture feasible and practical? If the answer is "yes", the result will be a fluid and flexible query processing system.

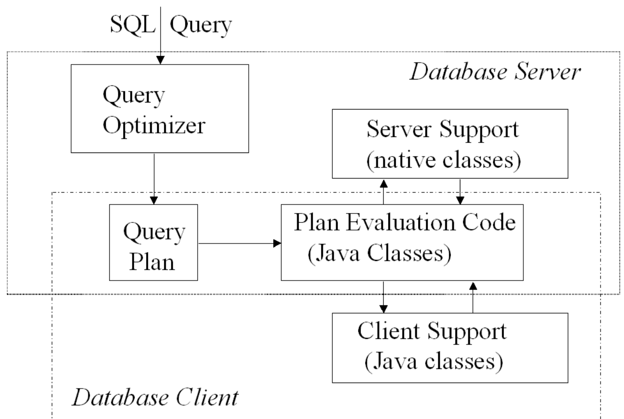

Figure 2: Design of Portable Query Processing

What is the ideal representation of query plan and plan evaluation code? The simplest approach is to translate the current C++ plan structure in PREDATOR into a Java data structure. In this case, the plan evaluation code corresponds to generic implementations of well-known relational operators (like nested-loops join, sort-merge join, etc.). What is the performance impact of implementing these algorithms in Java, as against in C++?

As a more general way of exploring the design choices, think of every query plan execution as requiring two components: a component that is specific to the particular query being executed (i.e. the query plan) and a component that is common for all queries (the plan evaluation code). There is no constraint that the query plan be a data structure. It could be a combination of a plan data structure and some specialized code to evaluate that particular plan. Our observation is that a "compiled" plan may be more efficient than simple data structures interpreted by generic plan evaluation classes. This is a well-known observation; however, the opportunity for such compilation will naturally arise because the query optimizer is creating Java data structures. It should therefore be relatively easy (in theory) to generate specialized Java classes to perform evaluation specific to this query. In the course of exploring this feature, we expect to encounter and address a variety of practical problems.

There is an interesting tradeoff between query-specific compiled code and generic plan evaluation code in a client-server environment. The goal of query-specific compilation is greater efficiency. However, if the code needs to be migrated to the client, there is an advantage to generic plan code that can be reused across queries.

In fact, one can treat the query optimizer as nothing but an SQL compiler. Typical compilers have a target "machine" language for their output. For traditional query optimizers, the target machine language is typically the relational algebra (at the "physical" level, where the implementation algorithms for each relational operator are also specified). We question the assumption that this is the appropriate target language, especially in an environment where plan evaluation classes need to be shipped to a client. Would a different choice of target language lead to smaller plan evaluation classes? From our experience building PREDATOR, we observe that many algorithms for relational algebra operators have similar structure and similar code sections. It is our surmise that there is a collection of primitive operations (at a lower level than the relational algebra) that can be combined to construct relational plans. There may be opportunities for traditional compiler optimizations (like invariant identification) in this target language. For example, consider the following query which is supported in PREDATOR: *SELECT * FROM A, B WHERE A.foo() = B.bar()*. If the query plan specifies a nested loops join with A as the outer table, A.foo() needs only be executed once for each distinct A tuple (although the predicate A.foo() = B.bar() is evaluated many more times). With the appropriate choice of primitives to represent this plan, it is easy to identify that A.foo() is invariant in the inner loop of the nested loops join.

5. Conclusion

We believe that the application of Java language technology to extensible object-relational database systems will lead to a novel database software architecture suitable for Web-based applications. Indeed, such Web-based systems are the next-generation of database systems. However, important research problems need to be addressed and this paper describes some of them. We are starting to explore these issues in the *Jaguar* project at Cornell.

6. References

1. *Shoring up Persistent Objects*: M.J.Carey, et al., Proceedings of the ACM SIGMOD Conference on Management of Data, 1994.

2. *The Case for Enhanced Abstract Data Types*: P.Seshadri, M.Livny. and R.Ramakrishnan, Proceedings of the Twenty Third International Conference on Very Large Databases, 1997.

3. *Optimization of Queries with User-Defined Predicates:* S.Chaudhuri and K.Shim, Proceedings of the Twenty Second International Conference on Very Large Databases, 1996

4. *Optimization and Execution Techniques for Queries with Expensive* Methods: J.M.Hellerstein, PhD Thesis, University of Wisconsin, 1995.

5. *Java-Relational Database Management Systems :* Cimarron Taylor. *http://www.jbdev.com/, 1997.*

6. *L.Haas, et.al. :* Starburst mid-flight: As the dust clears, IEEE Transactions on Knowledge and Data Engineering, March 1990.

7. *Application of Abstract Data Types and Abstract Indices to CAD DataBases :* M.Stonebraker, B.Rubenstein, and A.Guttman. *Proceedings of the Engineering Applications Stream of Database Week, San Jose, CA, May 1983.*

8. *Inclusion of New Types in Relational Data Base Systems:* Michael Stonebraker, Proceedings of the Second IEEE Conference on Data Engineering pgs 2*62--269, 1986.*

9. *Performance Tradeoffs for Client-Server Query Processing:* M.J.Franklin, B.T.Jonsson, D.Kossman, Proceedings of the ACM SIGMOD Conference on Management of Data, 1996.

10. *Client-Server Caching Revisited:* M.J. Franklin and M.Carey, Morgan Kaufmann, San Mateo, CA,1994, Editors: Ozsu, Valduriez, Dayal..

11. *JDBC Database Access with Java:* G.Hamilton, R.Cattell and M.Fisher, Addison Wesley, 1997.

12. *The Java Virtual Machine Specification:* T.Lindholm and F.Yellin, Addison Wesley, 1997.

13. *Implementing Multiple Protection Domains in Java*: T.von Eicken, C.Hawblitzel, C.Chang and G.Czajkowski, Proceedings of the Usenix Annual Technical Conference, June 1998.

14. *Secure and Portable Database Extensibility:* M.Godfrey,T.Mayr,P.Seshadri, and T.vonEicken, Proceedings of the ACM SIGMOD Conference on Management of Data, 1998.