Hybrid Shipping Architectures: A Survey

Ivan T. Bowman

University of Waterloo

itbowman@acm.org

11 February, 2001

Abstract

Recent advances in relational database systems include distributed systems that can choose to execute portions of query processing functionality at server or client sites. A symmetric problem that has received little attention is the partitioning of client application functionality between client and server. This report presents a survey of the literature related to both of these partitioning problems.

Contents

1 Introduction

Over the past few decades, relational database management systems (RDBMSs) have been widely adopted by business users to manage large amounts of corporate data. The adoption of the relational data model [7] owes its genesis to a fortunate convergence of several factors:

1. Concurrency controls and recovery mechanisms.
2. Access plan optimizers.
3. Efficient data processing algorithms.
4. The client-server computing paradigm.
5. Massive vendor marketing and sales efforts.

Although these factors have made RDBMS technology into a multi-billion dollar industry [33, 37], there are still several areas where this model can be substantially improved. In particular, it is still fairly difficult to write applications that use an RDBMS due to the *impedance mismatch* [8] between the set-based relational model and tuple-at-a-time procedural languages used to write applications. Further, these applications do not take the best advantage of global system resources. The clean separation of the system into a relational processor and procedural application code is a very convenient system architecture. The boundary between these components, however, enforces a partitioning that may not give the best possible performance.

The problem of under-utilized resources is exacerbated in the client-server model, where procedural application code typically executes on a separate machine from the RDBMS where the data is stored. In a pure implementation of this partitioning, the system is not able to exploit client resources to help compute the answers to relational queries, and can not use server resources to execute procedural application code.

To address these problems, researchers have proposed several ideas to better utilize client and server resources. The *hybrid shipping* model [11, 25] allows a relational processing system to execute portions of an access plan on a client site. User defined functions (UDFs) [13, 26] have been proposed to allow users to extend an RDBMS with procedural code that can execute either at the server or at the client. *Code-shipping* solutions have been proposed [29, 28] to allow optimizers to choose the best execution site for client application code. Related work in the area of distributed systems research has investigated mechanisms to automatically partition applications and optimize the placement of application code on multiple sites.

We are interested in a comprehensive system that achieves the following:

- Permits relational query processing to be performed at client and server sites.
- Automatically identifies fragments (at the granularity of basic blocks) of client application code that may benefit from being executed on the server.
- Provides a *safe* and *secure* mechanism for executing procedural client code in the database server.
- Optimizes the code placement of relational query processing operators and procedural client application code fragments to minimize either response time or resource consumption.

We hypothesize that this model will produce efficient access plans that yield optimal use of client and server resources. This execution model raises many issues which have been explored by other researchers. This paper presents a survey of the literature relating to these issues, and places this execution model in the context of previous research.

The remainder of this paper is organized as follows. Section 2 describes proposed mechanisms for partitioning the query processing effort of an RDBMS

between server and client machines. Section 3 discusses the symmetric problem of partitioning client application functionality between client and server machines. Section 4 describes approaches to optimizing queries in distributed systems. Section 5 describes approaches to implementing mobile code. Finally, Section 6 discusses how these topics fit together, and suggests areas for further research.

2 Partitioning Query Execution

While early database systems research concentrated on single-computer models, the advent of cheap, powerful workstations has led to systems that exploit the resources of multiple computers. Three orthogonal directions have been explored by researchers: *client-server systems* execute client applications on a separate machine from the RDBMS; *distributed database systems* use multiple server machines to answer queries; and *hybrid shipping* approaches allow the system to execute portions of a relational access plan on clients.

2.1 Centralized Systems

In the mid-1970's, several research and commercial prototypes of RDBMSs were created and evaluated; these prototypes were implemented with both the RDBMS and client application running on a single computer. System R [3] was developed by a group of about 15 researchers at IBM to run on IBM mainframes running the VM/370 operating system. INGRES was developed concurrently at the University of Berkeley [38] to run on PDP/11 computers running Unix. These systems were implemented with both the RDBMS and client applications residing on a single machine, with a relational interface between the two programs.

2.2 Client-Server Systems

One approach to partitioning functionality in a multiple-computer environment is to put client applications on a different computer from the RDBMS. Hagmann and Ferrari performed a performance study [16] with INGRES which suggested that performance could be improved by partitioning the system with the user interface and parser on one machine, and the rest of the RDBMS software on another machine.

The general form of such a partitioning approach is the *client-server* model, where there are many client machines running application software and some local RDBMS related software, submitting requests to a server machine which is running an RDBMS. This approach allows a system to scale more easily, since client resources are used to execute the client applications. Further, this approach has benefits for the administration of the system: only the server machines need to be backed up and administered, and client machines can be equipped with user interface hardware while server machines are equipped with large amounts of memory and large, reliable secondary storage.

The client-server architecture can be generalized to a multi-tier system, where computers are organized into layers with each layer acting as a client of the lower level and a server to the higher level. Even general peer-to-peer structures can be viewed as client-server, with each computer possibly acting as both client and server. For concreteness, we use client-server to refer to the case where each computer has a fixed role of either client or server.

Today, the majority of commercial RDBMSs (e.g. Sybase ASE, IBM DB2, Oracle, and Microsoft SQL Server) use some variant of the client-server architecture, where each site has a fixed role of either a client or a server.

2.3 Distributed Systems

As it became more common to have multiple computers available at a site, it made sense to use more than one computer to answer queries, and researchers considered several different ways to accomplish this partitioning. One partitioning was based on the idea of a *distributed RDBMS*: multiple computers were involved in the system, each providing a local RDBMS managing local data and contributing to a global distributed RDBMS. Several distributed RDBMS prototypes were implemented in the late 1970's, including R* [45], SDD-1 [4] and distributed INGRES [10]. These prototypes developed techniques for distributed query optimization and execution, distributed transactions, and distributed data (partitioning and replication). To some extent, these prototypes were ahead of their time, and failed to be widely accepted [37] for a number of reasons, including the lack of robust and efficient networking protocols in their operating systems.

More recently, newer distributed database systems have become very popular, with most commercial systems offering some form of distributed query processing [24]. There are also several research prototypes that provide distributed query execution, including Garlic [15, 30], DISCO [40], and Mariposa [34, 36].

2.4 Using Client Resources

One of the decisions that must be made when designing a client-server or distributed architecture is how to partition the query processing effort between the computers involved in the system. Traditional RDBMSs execute all of the query processing at servers, while object-oriented database management systems (OODBMSs) have also considered performing all of the query processing at the client site. A hybrid approach that dynamically chooses a combination of these alternatives has also been proposed.

2.4.1 Query Shipping

The *Query Shipping* approach is commonly used by commercial RDBMSs. This approach ships SQL queries from the client machines to the server. The server performs all of the query-processing effort, and the answer is returned in the form of a stream of tuples. Only a very limited form of caching can be used in this model: answers to previous queries can be stored in the client's cache to answer future identical queries. This model does not completely preclude query processing from occurring at the client site: some relational processing may be embedded in client applications in procedural code. For example, some client applications compute selections, aggregates, and even joins in customized procedural code. This approach is used by some data analysis programs which retrieve bulk data from a server and then perform a significant amount of analysis on the data solely on the client machine.

2.4.2 Data Shipping

In contrast, the *Data Shipping* approach does no query processing at the server. Instead, the query processing is all done at the client machines. As data is needed by client query processing algorithms, it is requested from the server. The data shipping approach makes much better use of client resources than query shipping, but it can also substantially increase communication costs; further, it can lead to under-utilization of server resources. Data shipping is primarily used by OODBMSs such as ObjectStore and O_2. A key decision for data shipping approaches is the unit of transfer of data [9]: the common choices are object servers (returning individual objects from the server), page servers (returning disk pages from the server without interpretation), and hybrid servers (dynamically choosing the better of the two shipping policies) [42]. Each of the object and page server approaches are preferable in certain conditions. Thus, the hybrid approach is likely to maximize performance since it can choose the better of either of the pure schemes

or some combination of object and page shipping.

2.4.3 Hybrid Shipping

An approach suggested by Franklin, Jónsson, and Kossman [12] is that of *hybrid shipping*. This approach allows the system to choose to perform some query processing on the server, and some portion of the processing on the client machine. Provided that an effective optimizer can be developed to choose the appropriate alternative, this approach has the ability to always perform at least as well as either of the two pure shipping approaches, and often much better. This approach requires that the client computer contains at least rudimentary query processing capabilities (the optimizer will only select plans supported by the client's query processor). Hybrid shipping allows the system to exploit client resources. In addition, the hybrid approach can substantially reduce communication overheads if data-inflating operations can be performed on the client instead of on the server. Further, the hybrid shipping approach can benefit from using client disk resources in parallel with server disk resources.

The hybrid approach is used in some database products such as UniSQL, application systems such as SAP R/3, and research prototypes such as ORION-2 [23]. This approach is also used by distributed wrapper based heterogenous systems such as DISCO [40], Garlic [15, 30], and Mariposa [34, 36].

The ADMS project [31] also uses hybrid shipping, although it requires that all aggregation, duplicate elimination, and sorting be done at the client during output generation, while joins and selections can be executed either at the server or at the client accessing cached results from previous queries.

2.5 Summary

While early systems were based on centralized execution, the current abundance of relatively inexpensive computer hardware connected by high-speed

networks has led to distributed systems being preferred for most installations. Commercial RDBMS systems predominantly offer pure client-server systems that employ query-shipping approaches. However, research prototypes and simulation studies have shown that these approaches lead to under-utilization of client resources. Instead, more flexible query execution policies are recommended that can partition data amongst several servers, and can choose to execute portions of query access plans on client computers. This approach provides a system with better scalability as the amount of data and clients increases. These approaches, however, only form part of the solution. They continue to require client applications to be executed exclusively on client computers. In Section 3, we discuss proposals that address this limitation.

3 Partitioning Client Applications

The hybrid shipping approach originally proposed by Franklin, Jónsson, and Kossman [12] considered partitioning the query processing operators. However, they do not address ways to partition the procedural client application code between client machines and servers. This is a capability that is provided by object-server OODBMS architectures [9] which allow methods to be executed at the server. In the case of selective methods, this approach can reduce the number of objects in intermediate results, leading to reduced execution and communication costs.

3.1 Automatic Partitioning

A similar problem occurred in the early 1970's when deciding how to partition functionality between a mainframe and *satellite* programmable graphical terminals. Programmers decided what functionality would be implemented on the satellite and which on the host, then implemented this functionality, usually in two different programming languages and environments. Two groups of researchers, Hamlin and Foley [17, 18] at North Carolina and van

Dam, Stabler, and Michel [41, 27] at Brown, found that this approach had several undesirable characteristics. They found that programmer intuition about where to place functionality was often wrong, especially in the early design stages when the code partitioning decision was usually made. The static nature of the partitioning decision also made it impossible to re-tune the application to work effectively under different work-loads or on a different host/satellite system with different cost parameters. Further, the approach required programmers to be fluent in two different programming environments. Combined, these problems made designing a distributed system for the host/satellite system substantially more difficult than a corresponding centralized system.

In order to address these problems, both groups developed automatic partitioning systems. Hamlin and Foley [18, 17] developed the CAGES system, which used a compile-time preprocessor to distribute the procedures of PL/I programs between a host and a satellite system, while van Dam, Stabler, and Michel [27, 41] developed the ICOPS system to partition programs written in Algol W. Both of these systems relied on run-time statistics to automatically partition the application procedures between the host and satellite. The CAGES project also considered moving global data, with a simple form of caching. These projects made it easier for application programmers to build efficient, working systems because they were able to treat the problem as if it were centralized, and they were able to defer partitioning decisions until run-time statistics could be gathered.

3.2 User Defined Functions

One possible way to partition client applications is to allow client programmers to define functions and operators that can subsequently be executed on the server. While some research has examined ways for client programmers to define new data types [35] and new access methods (such as Informix *data blades*), most commercial systems support extension through the use

of user defined functions. User defined functions are supported in 'universal' RDBMS systems such as Sybase ASE, Informix, DB2, and Oracle. For example, if a programmer wished to retrieve a list of all employees that had a street address that was a prime number, they could implement an `isPrime(n)` function and deploy it to the server. Subsequent queries could access this function in the WHERE clause to select the appropriate employees. User defined functions can also be used in the SELECT list of queries, and can also be used to compute user-defined aggregate functions. Users are required to decide what functionality should be executed at the server, then implement this functionality using the programming facilities of the RDBMS.

Developers of commercial RDBMSs are often wary of allowing client programmers to include code into their server due to the issues of *safety* and *security* [39]. Security refers to the threat that user defined functions may maliciously read or modify data that they should not have access to, or that they may pose denial of service attacks. Security issues are dealt with in commercial systems by requiring that only privileged users are able to install new user defined functions. The safety issue refers to the problem that bugs in user defined functions may affect the RDBMS; examples include causing a crash of the server, or causing undetected corruption of data.

The commercial implementations of user defined functions are not entirely satisfactory. Typically, they require either a proprietary interpreted language or access to a proprietary API to communicate with the RDBMS. In order to provide safety, some systems use interpreted functions at the expense of run-time efficiency. Other systems run native code in the server's process, but this leaves the server open to security and safety holes. The alternative of running native code in another process on the same computer as the server is also employed, also at the cost of run-time efficiency.

The use of Java as the source language has been proposed [13] as one alternative that could be used to provide limited safety with acceptable performance. The use of Java also promotes better portability, since access

to the RDBMS can be accomplished through a standardized API based on JDBC. Java UDFs can be developed and tested on client machines before they are deployed to servers for production use. While Java provides better support for portability, security, and safety, this comes at the cost of execution efficiency. Efficiency can be addressed to some extent using just-in-time compilation. Part of the cost, however, is the *impedance mismatch* between Java and the native language of the server. The JNI interface limits the efficiency of UDFs that make many callbacks to the server.

Another approach to providing safety is to use software fault isolation [43] which provides safety for native code. Wahbe, Lucco, and Andersen tested this approach with the POSTGRES user extensible type system. The software fault isolation approach provided significant performance improvements over using native operating system services and hardware address spaces to achieve safety.

Server-site user defined functions are installed at the RDBMS, and always execute there. In some cases, this is not desirable because it is less expensive to execute the UDF on the client site or because the UDF contains proprietary code that the application programmer does not wish to make available to the server. Mayr and Seshadri [26] address this concern by providing an optimization algorithm that accounts for the cost of executing a UDF at the client site, and allows concurrent execution to minimize the exposed network latency.

3.3 Stored Procedures

While UDFs are helpful in improving the execution speed of individual queries, they do not provide a general facility for performing data manipulation. Stored procedures are provided by RDBMSs as a mechanism for client programmers to group commonly used operations together and execute them efficiently on the server. For example, a common idiom in some programs is that of the 'guarded insert', which inserts a tuple if the corresponding key

does not exist, otherwise it generates an error message formatted using fields of the existing record. Without stored procedures, this requires two accesses to the RDBMS: one to find the tuple if it exists, and one to perform the insert. A stored procedure can reduce this to a single access.

In general, stored procedures can be arbitrarily complex. In fact, they can be used to execute arbitrary portions of a client application on the server, providing a form of execution partitioning. Unfortunately, the mechanisms for using stored procedures are cumbersome in similar ways to UDFs. The language or development environment of stored procedures likely does not match the environment of the client program, and any partitioning decision must be made early on in the design process.

3.4 Code Shipping

The user defined function model used in commercial systems and explored in the PREDATOR project does not consider moving functions to the server or client—the model assumes that partitioning is done statically by the application programmer. This requires that an administrator configure each site with the desired functionality.

This administration burden can become quite onerous as the number of sites grows into the tens, hundreds, or even thousands (quite achievable when every employee has a computer participating in the distributed system). The MOCHA project [29, 28] aims to ease this administrative burden by providing a *code shipping* middleware solution. This system allows access to a distributed group of heterogenous servers by using a wrapper architecture similar to Garlic. Unlike Garlic, the MOCHA wrappers (called data access providers, or DAPs) can be used to perform any of the operations supported by the system. If functionality is not present on the DAP, it is loaded dynamically by the system, by shipping the appropriate Java class files. Client programmers can integrate custom operators into this framework, and they will be likewise distributed by the system. This is accomplished by explicitly

describing meta-data for the operator using XML.

MOCHA optimizes queries using a dynamic programming approach [32] that builds left-deep processing trees. MOCHA optimizes to minimize communication costs by considering the *volume reduction factor*.

3.5 Summary

Relational database management systems allow client programmers to deploy portions of their application to the server machine in order to improve performance. The mechanisms provided, however, are cumbersome to use. They usually require that two languages and development environments be used, and also typically require that partitioning decisions be made at compile time. Because these mechanisms are so difficult to use, they are usually only attempted by experienced programmers. Current systems either rely on trusting these programmers to implement secure and safe extensions to the RDBMS, or accept a performance penalty to either interpret client extensions or execute them in an isolated process.

The programming language Java has provided some hope, in particular in the PREDATOR and MOCHA research prototypes. Java UDFs are portable, and the security and safety guarantees of the Java VM may be acceptable to RDBMS implementors. Java does come with a performance penalty compared to native code. The approach of software fault isolation offers another alternative, providing safe native code with low execution overhead.

Existing approaches to client program partitioning are fairly undeveloped, and require substantial effort and knowledge on the part of the application programmer in order to achieve an efficient, robust implementation. There is substantial room for progress in this area.

4 Optimizing Distributed Plans

The *query optimizer* component of an RDBMS chooses an *access plan* that specifies the operators that will be used to execute a query. There are two key components to the optimizer: the plan enumeration algorithm which uses heuristics to compare some subset of the possible execution strategies, and the cost function, which assigns a cost metric to each access plan.

For systems that permit operators to be executed at different sites, the system must provide an optimizer that chooses the site assignment in order to minimize some cost function.

The problem of optimizing queries has been shown to be NP hard. As a result, most systems rely on heuristics to prune the search space of possible plans. Because of this, the goal of the optimizer is usually viewed not as finding the *best* plan, but instead as eliminating all of the truly bad plans.

4.1 Search Space

For a centralized system, the following choices must be made by an optimizer:

1. Access path (e.g., index selection).
2. Join, aggregation, and duplicate elimination algorithms.
3. Join ordering.
4. Placement of expensive predicates.
5. Intra-query parallelism.

Distributed systems must decide all of the above for each of the server sites in the system. In addition, they must also make decisions on site selection. Further, the choices of join methods are expanded for distributed systems, where, for example, semi-joins may provide improved performance.

4.2 Costing of Plans

In order to compare two plans, an optimizer needs to assign them some cost function. In the original distributed RDBMS prototypes (R* [45], SDD-1 [4] and distributed INGRES [10]), this cost function was typically resource consumption (only distributed INGRES allowed plans to be optimized for response time). Further, communication costs were assumed to dominate any local processing costs, leading to fairly simplistic cost models.

Reasonably accurate cost models have been developed [14] for query processing algorithms. However, these cost models depend on several parameters that are often difficult to estimate, such as the selectivity of predicates. In particular, join predicates, sub-queries, and correlated conjunctive predicates are difficult to estimate accurately. Contemporary optimizers use statistics to guess at these parameters. Unfortunately, errors accumulate rapidly with the join degree of queries. Several researchers to proposed that query processing engines periodically evaluate whether the current plan is still believed to be optimal [14, 19].

4.3 Enumeration Strategy

The System R prototype used a dynamic-programming based optimizer [32] that discovered optimal left-deep processing strategies. Strategies for joining $n + 1$ tables were discovered using optimal join strategies for n tables. This enumeration technique has proven to be successful, and is emulated in many commercial RDBMSs. The original algorithm [32] only enumerated left-deep execution trees. This approach is acceptable in centralized systems, but distributed systems can particularly benefit from bushy trees that place adjacent operators on the same site. Modifications of the dynamic programming approach allow bushy trees to also be enumerated at the expense of greater optimization cost. A more serious criticism of the dynamic programming algorithm is the fact that it uses exponential time and space; this limits

the optimization algorithm to joins of about fifteen quantifiers [24].

Greedy algorithms use heuristics to prune the search space, and can tolerate much higher join degrees at the cost of possibly poorer access plans. Alternative branch-and-bound style algorithms [5] can generate reasonable plans with quite low memory consumption, allowing optimization of queries with over 80 quantifiers.

4.4 Optimizing Expensive Predicates

Traditional query optimizers are primarily concerned with choosing an optimal join ordering and appropriate join methods. When user defined functions are also included, the optimization problem becomes substantially harder. Estimation of selectivity and cost of user defined functions is non-trivial, although a reasonable guess may be accomplished by maintaining statistics of previous executions. Some researchers have proposed that the placement of expensive predicates such as UDFs be considered during the join enumeration algorithm [6, 20]. This approach has the effect of substantially increasing the cost of enumeration.

Hellerstein proposed a predicate migration approach [20], which can lead to cost that is polynomial in the number of UDFs. In the worst case, predicate migration requires exhaustive enumeration of the join space, with cost $O(n!)$ in n the number of joins in the query. Despite this possibly high cost, predicate migration can not guarantee optimality of the resulting access plan, even with the restriction of a linear cost function.

The approach of Chaudhuri and Shim is polynomial in the number of user-defined predicates, never needs to exhaustively enumerate all join strategies, and guarantees the optimality of the resulting access plan. Like the predicate migration approach, this approach is sensitive to the estimated selectivity of user-defined predicates. Very poor plans may be chosen if these estimates are not accurate.

Both expensive predicate placement algorithms were originally designed

in the context of a centralized system, and do not account for the possibility of selecting an execution site for expensive predicates. Mayr and Pirahesh [26] consider the location of user predicates to the extent that client-site UDFs are required to be located at the client site. Their join enumeration algorithm is exponential in the sum of the join degree of the query and the number of UDFs present in the query.

The cost of enumeration is worse if the optimizer also considers the best site at which to execute UDFs. For queries with fairly few UDFs, a dynamic programming approach may be reasonable. For queries that contain tens of UDFs or a high join degree, heuristic approach such as greedy algorithms, randomized algorithms, or branch and bound algorithms [5] will be required, since dynamic programming will not be able to optimize such queries in a reasonable amount of time.

For very large queries, for example those generated by considering all basic blocks of a client application as candidates for UDFs, existing join enumeration algorithms are unlikely to give reasonable results. Alternative approaches based on commodity flow networks [22] may prove to be fruitful in such situations.

4.5 Summary

Relational query optimizers use heuristics and cost estimates to generate access plans which specify the access methods to use for base data, join ordering and methods, and placement of expensive predicates. In the distributed case, optimizers also choose the execution site for each of the operations in the access plan. Some optimizers consider the placement of expensive predicates (UDFs) during join enumeration. Although Chaudhuri and Shim's approach is polynomial in the number of UDFs, their approach does not cover the distributed case where UDFs can be executed at the client or server, as specified by the optimizer. This case is likely to be substantially more expensive, and may rule out the use of traditional dynamic programming algorithms for

optimization.

5 Mobile Code

The problem of executing client application code on a RDBMS server can be viewed as a special case of mobile code [39]. There has been a recent upsurge in interest in mobile code environments, partially fueled by the Internet and the introduction of the Java programming environment.

In general, the problem of writing a distributed application is substantially harder than writing an equivalent centralized application. Waldo et al. [44] argue that this is an inherent problem of distributed computing that can not be "papered over" by syntactic sugar. With this caveat in mind, we examine proposed systems that attempt to ease the burden of creating efficient, robust distributed applications.

5.1 Automatic Distributed Partitioning

One significant problem that occurs when writing a distributed system is the decision of how to partition the functionality. As discussed in Section 3.1, previous researchers have found that programmer intuition is often wrong when it comes to choosing a partitioning. These researchers propose using an automated partitioning system which examines a system written for a single computer, identifies portions which can be distributed, and chooses a partitioning based on profiling statistics.

The ICOPS [27, 41] system provided automated distributed partitioning of Algol W programs between a host and satellite computer. Procedures could be re-partitioned at run-time based on measured statistics.

In contrast, the CAGES [18, 17] allowed compile-time partitioning of PL/I procedures between the host and satellite. No automatic partitioning was offered, but a *nearness matrix* was provided as an aid to the programmer. A significant advantage of the CAGES system was its ability to execute

code at the host and satellite concurrently, albeit only in a fairly constrained way. In one performance study, the CAGES project found that an application, MEDICS, could not be efficiently partitioned because the procedures all contained both graphical display routines and expensive computation. Since the granularity of distribution was procedures, the system was not able to move the computation to the host and display routines to the satellite until the authors re-wrote the procedures to allow partitioning.

Both ICOPS and CAGES only considered partitioning on the order of two dozen modules. Large modern programs contain hundreds or thousands of procedures. The Coign project implemented by Hunt [21] was able to partition large applications containing hundreds of COM components among distributed computers. Coign uses system monitoring to gather statistics for an optimization step. Optimization is performed using a classification phase that clusters components with similar access behaviour, and a graph cutting algorithm using a commodity flow network. The Coign system relies on finding components that can be effectively deployed. In studying the PhotoDraw system, Hunt found that Coign was severely constrained by the large number of non-remotable components.

Based on the observations of Coign with the PhotoDraw system and CAGES with the MEDICS system, it appears that Waldo et al.'s warning [44] is particularly pertinent: it is not trivial to generate a robust, efficient distributed system from a system that was intended to be centralized.

The Abacus project [1] does not attempt to distributed programs that were intended to be centralized. Instead, it provides primitives for programmers to write applications from data-intensive, distributable components. The system monitors run-time resource usage, and dynamically changes component placement using checkpoint and restore methods explicitly coded by the programmer. This system manages to adapt to changing work loads. An interesting feature from the RDBMS standpoint is that, since the partitioning is done dynamically, it is resistant to bad initial access plans.

5.2 Languages

'Safe' languages hold out the promise that bugs in a program written in a safe language will not affect unrelated programs in the environment. Further, mobile code benefits from languages that can help enforce security policies.

Java [2] has recently become a very popular language for mobile code. It provides portability by using a platform-neutral byte-code representation and a rich, standardized class library. The security model in Java addresses at least some of the security and safety concerns related to mobile code [13], and the emergence of just-in-time compilers (JITs) holds out hope that the execution gap between Java and systems-level languages such as C will narrow.

However, Java is not the only approach to native code. Objective Caml, Obliq, and Safe Tcl [39] are only three of the alternatives.

In addition to using a custom language, a technique such as software fault isolation [43] can be incorporated into the compiler for existing languages such as C and Pascal. This approach inserts verification code around possibly unsafe operations such as array accesses, and uses optimization techniques to reduce the overhead of these checks.

In the context of RDBMSs, Java has been used in MOCHA and PREDATOR both to implement query operators on client machines and to implement UDFs to execute on the server. Although Java is an expedient choice for such uses, it is not clear that it actually provides the best level of abstraction. It may be the case that a programming language with primitives more closely matched to relational query processing would provide a better match. Such a customized approach, however, would require substantially more implementation effort compared to using an off-the-shelf language.

6 Conclusions

Relational database management systems have progressed to the point where we are now interested in distributed systems of multiple servers and many clients. We know that the best performance is achieved if we consider executing relational query processing both at the server and at the client, and there are several proposals for optimizing plans for such systems and ensuring that the appropriate operator code is available. Further, we know that client applications can benefit from executing portions of their procedural code on servers by using such mechanisms as user defined functions and stored procedures.

There is, however, a significant lack of research in the direction of automatically distributing portions of the procedural client application for execution on server machines. Such a facility is needed, because not only is it difficult for programmers to implement the partitioning due to cumbersome server interfaces, but also because programmers often make sub-optimal decisions that can not be easily corrected after applications have been deployed.

In order to allow more effective utilization of client and server resources, we should research the following topics:

1. How can we *automatically* detect sub-procedural fragments of a client application that may benefit from executing at the server?

2. How can we optimize the placement of a very large number of UDFs and stored procedures between a client and server?

3. How can we execute client code on a server efficiently, safely, and securely?

Answers to these questions will surely help us to build more efficient distributed database systems.

References

[1] Khalil Amiri, David Petrou, Greg Ganger, and Garth Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Sun Microsystems, 1996.

[3] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K.P.Esqaran, J.N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 2(1):97–137, 1976.

[4] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 6(4):602–625, 1981.

[5] Ivan T. Bowman and G. N. Paulley. Join enumeration in a memory constrained environment. In *Proc. IEEE Conf. on Data Engineering*, 2000.

[6] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. In *Proc. Int'l Conf. on VLDB*, Mumbai(Bombay), India, 1996.

[7] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[8] George P. Copeland and David Maier. Making smalltalk a database system. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting*, pages 316–325, Boston, Massachusetts, 18–21 June 1984. ACM Press.

[9] David J. DeWitt, Philippe Futtersack, David Maier, and Fernando Vélez. A study of three alternative workstation-server architectures for object oriented database systems. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *Proc. Int'l Conf. on VLDB*, pages 107–121, Brisbane, Australia, 13–16 August 1990. Morgan Kaufmann Publishers.

[10] R. Epstein et al. Distributed query processing in a relational database system. In *Proc. ACM SIGMOD Conference*, May 1978.

[11] Michael J. Franklin and Michael J. Carey. Client-server caching revisited. In M. Tamer Özsu, U. Dayal, and Patrick Valduriez, editors, *Distributed Object Management*. Morgan-Kaufmann Publ. Co., San Mateo, CA, USA, May 1994. International Workshop on Distributed Object Management.

[12] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proc. ACM SIGMOD Conference*, pages 149–160, Montreal, Quebec, Canada, 4–6 June 1996. ACM Press.

[13] Michael Godfrey, Tobias Mayr, Praveen Seshadri, and Thorsten von Eicken. Secure and portable database extensibility. In Laura M. Haas and Ashutosh Tiwary, editors, *Proc. ACM SIGMOD Conference*, pages 390–401, Seattle, Washington, USA, 2–4 June 1998. ACM Press.

[14] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[15] Laura M. Haas, Donald Kossman, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proc. Int'l Conf. on VLDB*, Athens, Greece, 1997.

[16] Robert Brian Hagmann and Domenico Ferrari. Performance analysis of several back-end database architectures. *ACM Transactions on Database Systems*, 11(1):1–26, March 1986.

[17] Griffith Hamlin, Jr. Configurable applications for satellite graphics. In *Proceedings of the Third Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 76)*, pages 196–203, Philadelphia, PA, July 1976. ACM.

[18] Griffith Hamlin, Jr. and James D. Foley. Configurable applications for satellite graphics (cages). In *Proceedings of the Second Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 75)*, pages 9–19, Bowling Green, Ohio, June 1975. ACM.

[19] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 2000. See `http://telegraph.cs.berkeley.edu/`.

[20] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. ACM SIGMOD Conference*, pages 267–276, Washington, DC, 1993.

[21] Galen C. Hunt. *Automatic Distributed Partitioning of Component-Based Applications*. PhD thesis, University of Rochester, August 1998. Also available as Technical Report TR695.

[22] Galen C. Hunt and Michael Scott. The coign automatic distributed partitioning system. In *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI '99)*, pages 187–200, New Orleans, LA, February 1999. USENIX.

[23] B. Paul Jenq, Darrel Woelk, Won Kim, and Wan-Lik Lee. Query processing in distributed orion. In F. Bancilhon, C. Thanos, and D. Tsicritzis, editors, *Advances in Database Technology–EDBT'90*, pages 169–187, Venice, Italy, March 1990.

[24] Donald Kossmann. The state of the art of distributed query processing. *ACM Computing Surveys*, 2001. To appear.

[25] Donald Kossmann and Michael J. Franklin. A study of query execution strategies for client-server database systems. In *Proc. ACM SIGMOD Conference*, 1995. Also available as University of Maryland Technical Report CS-TR-3512.

[26] Tobias Mayr and Praveen Seshadri. Client-site query extensions. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proc. ACM SIGMOD Conference*, pages 347–358, Philadephia, Pennsylvania, USA, 1–3 June 1999. ACM Press.

[27] Janet Michel and Andries van Dam. Distributed processing on a host/satellite graphics system. In *Proceedings of the Third Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 76)*, pages 190–195, Philadelphia, PA, July 1976. ACM.

[28] Manuel Rodríguez-Martínez and Nick Roussopoulos. Automatic deployment of application-specific metadata and code in MOCHA. In *Proc. 7th EDBT Conference*, Konstanz, Germany, March 2000.

[29] Manuel Rodríguez-Martínez and Nick Roussopoulos. MOCHA: A self-extensible database middleware system for distributed data sources. In *Proc. ACM SIGMOD Conference*, Dallas, Texas, USA, May 2000.

[30] Mary Tork Roth and Peter Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. Int'l Conf. on VLDB*, Athens, Greece, 1997.

[31] Nick Roussopoulos, Chungmin M. Chen, Stephen Kelley, Alexios Delis, and Yannis Papakonstantinou. The ADMS project: Views "R" us. *Bulletin of the Technical Committee on Data Engineering*, 18(2):19–28, June 1995.

[32] P. Griffith Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Conference*, 1979.

[33] Avi Silberschatz and Stan Zdonik et al. Strategic directions in database systems – breaking out of the box. *ACM Computing Surveys*, 28(4):764–788, 1996.

[34] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A new architecture for distributed data. In *Proc. IEEE Conf. on Data Engineering*, pages 54–65, Houston, TX, 1994.

[35] Michael Stonebraker. Inclusion of new types in relational database systems. In *Proc. IEEE Conf. on Data Engineering*, pages 262–269, Los Angelese, CA, February 1986.

[36] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[37] Michael Stonebraker and Joseph M. Hellerstein, editors. *Readings in Database Systems*. Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[38] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, 1976.

[39] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3), September 1997.

[40] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5), September/October 1998.

[41] Andries van Dam, George M. Stabler, and Richard J. Harrington. Intelligent satellites for interactive graphics. *Proceedings of the IEEE*, 62(4):483–492, 1974.

[42] Kaladhar Voruganti, M. Tamer Özsu, and Ronald C. Unrau. An adaptive hybrid server architecture for client caching ODBMSs. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proc. Int'l Conf. on VLDB*, pages 150–161, Edinburgh, Scotland, UK, 7–10 September 1999. Morgan Kaufmann.

[43] Robert Wahbe, Steven Lucco, Thomas E. Andersen, and Susan L. Graham. Efficient software based fault isolation. In *Proceedings of the Symposium on Operating System Principles*, 1993.

[44] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., Mountainview, CA, November 1994.

[45] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. R*: An overview of the architecture. Technical Report RJ3325, IBM Research Lab, San Jose, CA, 1981.