TECHNICAL RESEARCH REPORT

An Architecture for a Mobile and Dynamically Extensible
Distributed DBMS

by M. Rodriguez-Martinez, N. Roussopoulos

CSHCN T.R. 98-2
(ISR T.R. 98-10)

The Center for Satellite and Hybrid Communication Networks is a NASA-sponsored Commercial Space Center also supported by the Department of Defense (DOD), industry, the State of Maryland, the University of Maryland and the Institute for Systems Research. This document is a technical report in the CSHCN series originating at the University of Maryland.

Web site http://www.isr.umd.edu/CSHCN/

An Architecture for a Mobile and Dynamically Extensible Distributed DBMS

Manuel Rodríguez-Martínez Nick Roussopoulos
manuel@cs.umd.edu nick@cs.umd.edu
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland at College Park

1 Purpose

This report proposes an architectural framework for the design and implementation of a Distributed Database Management Systems which integrates Java and the Java Runtime Environment with the static set of operations found in a traditional database execution engine. With such system, we intend to study the benefits of mobility of DBMS code (*functionality shipping*). This functionality shipping occurrs at several levels, namely: query (i.e. predicates), executor (i.e. join operator) and access methods (i.e. index scan) levels.

2 Introduction

Traditionally, a DBMS has been implemented as a monolithic system giving users a predifined set of operations and data types. The DBMS internals are closed to third parties, forcing enterprises to wait for the DBMS vendor to release new versions of the code that include new features and enhancements to the system. Often, developers at the enterprise cannot take advantege of tools distributed by third-party vendors since these require some application programming interface that many DBMSs do not include. This scheme makes software development expensive since developers are tied to the policies and tools provided by the vendor of the DBMS. There are some systems such as Postgres ([SK91]) and Informix ([Sto97]) which enable users to define new datatypes and functions. However, much of the DBMS internals are still closed and the new functionality if often restricted to be used in limited areas of the system (i.e. define new predicate functions for queries).

These problems just mentioned have also been faced by the developers of other types of applications such as word processors, spreadsheets, distributed computing tools, etc. Many of them are now focusing their attention on the use of the Java programming language ([GM95]) as a tool to somewhat alleviate some of these problems. Their interest comes from the fact that Java provides an object-oriented and architecture independent programming paradigm which allows developers to create modular applications that can be run in any *Java-enabled* computer.

Using the object-oriented paradigm and software engineering techniques, developers are exploring new ways to build applications which are not monolithic. As a result, many developers are now using a componet-based approach where an application is built by combining several independent software components. The code in each component implements some part of the functionality needed by the targeted application. Typically, a component is made out of one or more classes, each one containing the datatypes and methods needed to carry out a specific function. A well defined application programming interface for exporting and importing functionality between components is defined and standardize. An advantage of this approach is the modularity and reusability of the code been written, and the ability to change parts of an application in a *plug-and-play* manner. It also enables the inclusion of third party software components so as to use the most suitable tools available. Another advantage is the ability to make applications interact with each other and with the host operating system is a easy and efficient manner by using the proper component interface.

With the full support of classes in Java, developers are exploiting all the features mentioned above and in addition, they are creating just one version of the software that can

be tested at many different machines. For example, a word processor can built, possibly reusing some of the code from an existing spreadsheet application, and then run in several platforms without the need for recompiling it as it would be the case if C or C++ where used instead. Some developers, such as Corel, are even converting their applications into applets, which are pieces of code that can be run in any Java-enabled Web browser (i.e. Netscape Navigator). It can be argued that Java's popularity is most likely due to the fact that it came at a moment when system administrators and developers where looking for ways to cut down the cost and complexity of multi-platform application development and maintenance.

Given these facts , there are several questions that need to be raised in the context of distributed database systems. Can we develop a DBMS where some of its internal modules (i.e. optimizer, query execution engine, etc.) can be tailored by system programmers or third-party developers, using say Java, to fit the changing needs of the enterprise by adding or removing some components? Will it be possible to ship functionality (i.e. Java code) between two DBMS in a Distributed Environment to compensate for the lack of some operations at one or both sites? Can we perform this code shippment dynamically? What is the kind of functionality that should be shipped? What is the cost model needed to determine when such code shipping is to be perform? What kind of application programming interface is needed to allow a component from one site to migrate to another and work properly? Which assumptions can be made by the query optimizer about the system? To what extent does this approach solves some of the problems present in heterogeneous database environments?

The architecture been proposed in this report attempts to provide some foundation over which research can be carried out to try to answer some of the question mentioned above. In this report, the phrases code mobility and functinality shipping are used interchangebly. The organization of the remainder of the report is a follows. Section 3will give a brief survey about some of the work done in centralized and distributed databases. Section 4 gives the formulation of the problem that is to be solved and a motivating example is discussed. In section 5 our architecture is presented in detail with the features it provides and its possible limitations. Finally, section 6 gives a summary and presents options for the future work needed to implement and evaluate a prototype system based on the architecture been proposed.

3 Related Work

Much of the work in centralized and distributed databases could be classified upon the assumptions made by each system in terms of the query processing capabilities , system configuration and extensibility of the DBMS used at each site. In the discussion that follows we mainly use a relational and object-relational terminology.

In terms of query processing, most distributed systems have used either a *query shipping* or *data shipping* approach ([SAD+94]). In query shipping, a query is sent to and processed by the machines which maintain the tables that are referenced in the query. Once the operation has been completed, the target machines return the results to the source which issued the original request for the query. Query shipping puts most of the load on the database server machines. Only requests and results (final or partial) are sent accross the network connecting the machines in the system. The resources at the machine issuing the query

are only used for data display and hence might be under utilized. Data shipping has been mostly used by distributed relational or object-relational systems such as R* ([WDH+81]), ADMS ([RCK+95]) and Mariposa ([SAP+96]).

Data shipping moves all the data referenced in a query from the data sources to the machine from which the query was issued. In this approach, most of the load is put on the client machine (where the query was issued) and the network connecting it to the other members of the system. The data sources are only used for the purpose of retrieving the data and therefore, their resources cannot be exploited for the benefit of query processing. Distributed Object-Oriented Databases typically employ the data shipping approach. In [FJK96], a *hybrid shipping* approach is proposed which combines both data and query shipping. With this approach, the machines in the system might perform either query or data shipping depending upon the benefits that each approach might have for answering the query at hand. The rationale behind hybrid shipping is the possible minimization of network traffic (as in query shipping) combined with the ability to move the data (as in data shipping) to machines with plenty of resources (i.e. CPU, memory, disk). The simulator used in [FJK96] and the Tornadito DBMS prototype ([PP96]) are examples of systems using the hybrid shipping approach.

Another way to classify distributed systems is based on the type of environment been assumed to operate at each site. In an *homogeneous environment* every site that forms part of the distributed system is assumed to run the same copy of the DBMS and have the same hardware configuration. This approach gives a framework for the desing of relatively easy cost models for query processing and to carry out DBMS performance evaluation studies. It also makes the interoperation between different sites easy, since there is a well-known set of operations and data types present everywhere. This type of environment is most likely to be found within the confines of the Local-Area Network (LAN) of a particular enterprise. Some systems assuming this type of environment are R* ([ML86]), ADMS ([RCK+95], [DR92]) and DIMSUM ([FJK96]). Although, Mariposa ([SAD+94], [SAP+96]) does not assumes hardware homogeneity, it does assumes DBMS homogeneity since each site runs the same copy of the Mariposa DBMS.

The *heterogeneous environments* are composed of sites running different types of DBMSs. There might be sites that do not have a DBMS at all, only exporting flat files to the rest of the system. The hardware used at each site is different too, possibly ranging from personal computers to mainframes. Query processing is much more complex in this type of environment. Each site as a different representation for the data, different types of operations and sometimes, the semantics of an operation at one site are different from thatof the same operation at another site. This type of environment is more common than an homogenous one and can be found in both Local-Area Networks (LANs) and Wide-Area Networks (WANs). Typically, a middleware software layer is used to achieve interoperability among the data sites. This middleware is mainly composed of *wrappers* and *mediators* as shown in figure 1 on page 4. A wrapper translates the data stored at a given source to a global format (schema) that can be understood by the mediators. These mediators are the entities that perform distributed query processing in the system. Systems such as DISCO ([TRV96]), Garlic([RS97]) and TSIMMIS([CGMH+94]) use this type of mediator-wrapper architecture to handle data in heterogeneous environments.

Finally, we can classify a DBMS as been *static* or *extensible*. A static DBMS gives

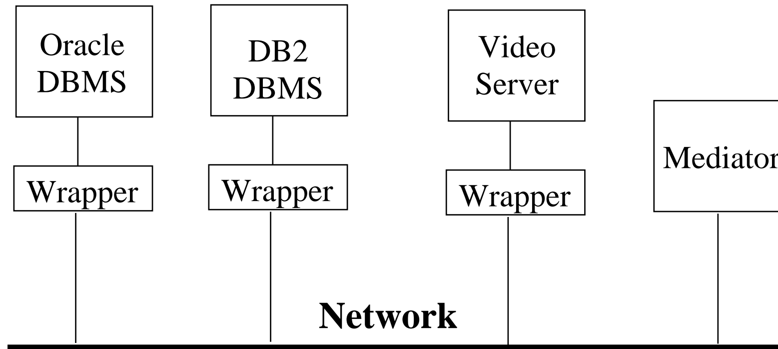

Figure 1: Wrapper-Mediator architecture.

the users a predefined set of operations and data types. The users need to map the data types and the operations that naturally arise in their problem to the those of the DBMS. Often, this mapping is not easy or impossible to do, thus making the implementation of an application complex or not as powerful as needed. Examples of systems which are static in nature are System R ([Ae76]), R*, ADMS and INGRES ([SWKH76]).

An extensible DBMS provides the user with the ability to add new functions and data types to the system by writting them using a progamming language such as C or C++. These new types and functions are then registered inside the DBMS and are used by it as if they were built-in ones. With these new tools, an easier and cleaner application can be developed. Also, the DBMS is enhanced with new features that were not present in the original distribution copy. Furthermore, a rule system might also be included within the DBMS, with which system behavior can be modified by adding or removing the proper rules. Postgres, Informix, PREDATOR ([SP97]), and most Object-Oriented DBMSs are cases of extensible systems.

The architecture we propose envisions a system where parts of a DBMS can migrate to another DBMS. This movement will make the receiving DBMS to be dynamically extended. There will be no need for the users to manually move the code, since code mobility will occur automatically. We this power, problems with a heterogenous source might be solved by simply giving the operation that we whish to perform to the DBMS running at the source. We can also avoid performing large data transfers to machines which are the only ones known to carry out an specific function. By simply shipping the code for the operation to the data source and execute the operation there, considerable network bandwith can be saved. Of course, this will only be done if the size of the code is much smaller than the size of the data. Ideas such dynamic load balancing, networks of PCs running parts of the DBMS and dissemination of DBMS components to mobile computers can be explored using the framework our architecture will provide. These are the some of the features that make our approach different from the ones taken by the systems mentioned in this section. A more detailed formulation of the ideas that motivate our system follows.

4 Problem Formulation

The problem to be solved is the design and evaluation of an architecture which includes policies, algorithms, cost models, data types, utility libraries and application programming interfaces that can be used to implement a distributed database system where *hybrid shipping*, *dynamic extensibility* and *dynamic code mobility (functionality shipping)* between sites is possible. This dynamic code mobility can occur at any of the following levels:

- **Query Level** - functions used as predicates in queries, aggregate operators, etc., should be able to migrate to a remote site and execute properly. Similarly, the code used to create, materialize and maintain views should be able to migrate too.
- **Executor Level** - the code used to execute relational operators such as selection, projection and join might move to a site where one or more of these operators do not exist or do not implements the variant that is needed (i.e. ship code for hash join).
- **Access Methods Level** - it is possible to migrate code which creates and maintains indices such B-trees, R-trees and Grid-files.
- **Query Optimizer Level** - the code needed to compute the cost of the functions, operators and access methods should move as well.

Notice that dynamic extensibility is achieved when **mobile code** arrives from other DBMS. A programming language such as Java can be used to implement all this mobile code. Thefore, the architecture should include a mechanism to integrate Java with the rest of the DBMS which might have been written in some other programming language, most likely C. This the approach that our architecture will follow.

We illustrate the power of a system with the characteristics just mentioned above with an example of the possible ways in which a selection operation can be performed. Suppose that there is a table storing information about car parts at a warehouse located in San Francisco. The parts table has the following schema: *sf_parts(number, name, quantity, price)*. Fields *number* and *name* are srtings giving the part's identification number and name, respectively. Field *quantity*, an integer, gives the number of parts currently in stock. The last field, *price*, is a real number representing the price of the part in dollars. The table as 30,000 entries, each having a size of 100 bytes. The user of a DBMS running at the headquarters of parts manufacturer Y issues the following query to her local DBMS:

```
SELECT *
FROM sf_parts
WHERE we_make_it(name)
```

The headquarters of manufactures Y are located in New York. Function `we_make_it` takes the part's name and tries to match it to one of the names of parts known to be made by manufacturer Y. It returns true if a match is found or false otherwise, and it is a user-defined function written by a system programmer working for Y. Around 15% of the parts stored in *sf_parts* are made by Y. The bandwidth of the network connecting both sites is 1.5 Mbits/sec (a T1 link).

There are two choices to evaluates this query:

- Ship all the tuples from San Francisco to New York, then run the `we_make_it` predicate at New York. This requires moving 3 MB worth of data across a Wide-Area Network.
- Use `we_make_it` as a filter by running it at San Francisco and then sending only those tuples which satisfy the predicate. This will reduce the amount of data sent accross the network.

For the first stategy, moving all 3 MB of data to New York will take 16 seconds. The second strategy only moves 450 KB of data to New York, taking just 2.4 seconds. Assuming the network bandwidth is the critical factor, the second strategy should be the preferred one. Unfortunately, to achieve this strategy in current systems, the user will have to access the DBMS at San Francisco, determine if it is possible to add new functions to the it, copy and compile `we_make_it` there, before even running the query. This process will have to be repeated for each place at which the predicate needs to be run. However, a distributed system that can ship code from one site to another, might simply have the DBMS at New York send the code that implements function `we_make_it` over to the DBMS at San Francisco automatically.

In addition to moving predicates, we could move code that implements relational operators, maintains indices or performs any other DBMS operation. For example, one can have ship the code for a hash join and use it, instead of nested loops join. These are the kind of problems that the architecture proposed in this report attempts to solve. The next section describes the main components of our architecture.

5 Mobile and Dynamically Extensible DBMS Architecture

The main new feature of the system that we propose is the ability to dynamically move components from one DBMS to another. This ability will be used to help compensate for the lack of functionality at the DBMS where the code is been shipped. As mentioned in section 3, query shipping moves the query to the data source, while data shipping moves the data to the query. Our architecture introduces a new feature: the DBMS goes to the data source.

Figure 2 on page 7 presents the general architecture of the system we envision. Each site that is part of the distributed database has at least one the following processes:

- **C: Client Application** - submits queries from the user into the system and displays the results obtained from these back to the user.
- **MW: Middleware Process** - processes the queries issued by the clients and contacts the necessary data sources to coordinate the processing of the query. Operations such as query parsing, optimization and parallelization occur at this process.
- **B: Backend Process** - executes the operations that the middleware sends in order to evalute the query. Operations such as relation scans, joins, projections and predicate evaluation occur at this process. The results of such operations might be send

to a middleware process or to another backend process to continue further with the execution of the query.

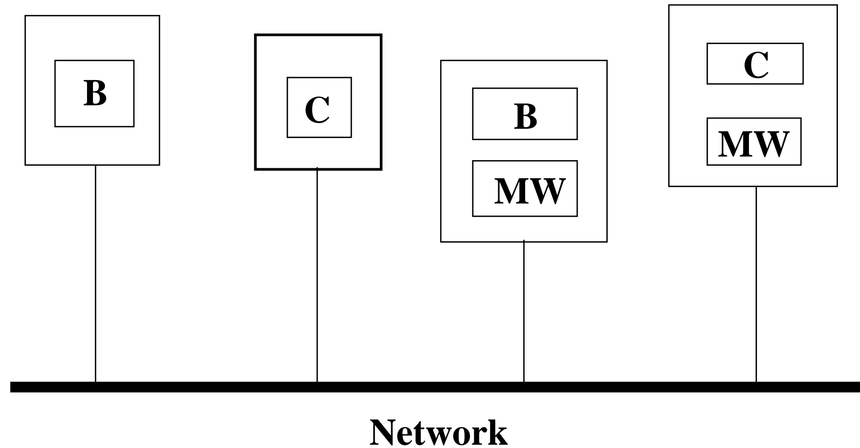

Figure 2: General System Archicture.

The configuration of our system is very similar to that used by the Mariposa DBMS. However, Mariposa does not supports code mibility as ours does. At the Middleware and Backend processes, there exists a **Java Runtime Execution Environment** which is used to execute the mobile DBMS code. This mobile code is received as part of the query evaluation and execution plan. Alternatively, the plan might include directons as to where does the code can be found.

Our desing does not assumes that the same hardware and/or DBMS is used at every site. We expect to be able to handle different types of machines such as workstations, network computers, multiprocessor systems and mainframes. Data sources might be running entirely different DBMS (DB2 and Informix) or different releases of the same system (i.e. Oracle 7 and Oracle 8). However, we assume that the DBMS at each site conforms to the architecture used by the Middleware and the Backend components that we propose. Notice that this assumption is very is similar to the one made by developers of Web pages which contains applets. There is no assumption about which specific Web browser will be used to download and read the Web page. All that is assumed is that the Web browser conforms to an architecture where there is a Java Runtime System[1]inside the browser that will handle the applet's code. In our case, we assume that there will be a Java Runtime Execution Environment that will take care of handling the mobile code. The interface necessary to support this behavior is described in section 5.

To integrate *legacy data sources*[2] into the distributed system, we propose the use of a modified version of the Backend process that works as a wrapper but , unlike most wrappers, it also has query execution capabilities. This is a departure from the traditional wrapper-mediator architecture presented in section 3. The Backend is able to translate data from

[1]This also known as the Java Virtual Machine.

[2]These are sources running a DBMS which does not conforms with our architecture or ones where there is no DBMS at all (i.e. a Web server). For this type of systems the integration of a Java Runtime Execution Environment could be difficult or impossible to achieve.

the legacy system to the schema used in the distributed system, much like a wrapper does. But in addition, it can execute many of the operations (i.e. select, project and join) that a traditional DBMS or a mediator does. In this scheme, the role of the wrapper is completely assumed by the Backend, while the functionality of the mediator in split between the Middleware process and the Backend process. A more detailed description of this scheme is presented in section 5.4.

We now present a general description of the structure and functionality that can be found in the Middleware Process, the Backend Process and the Java Runtime Execution Environment. The main goal of this description is to show how a Java Runtime System (Java Virtual Machine) can be integrated into an existing DBMS system. When such integration is not possible, the functionality can be added as a component outside the DBMS. Section 5.4 shows how this external component can be structured to interact with the DBMS.

5.1 Middleware Level

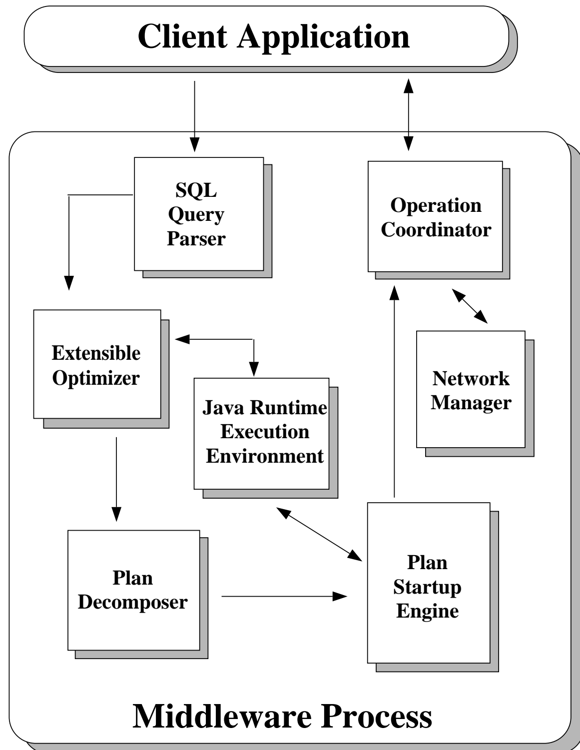

Figure 3: Middleware component.

The *Middleware* component provides client applications with a point of access to the distributed system. Queries submitted by client applications are analized and prepared for execution by this component. Once a query is ready for execution, the Middleware contacts the necessary data sources and coordinates their efforts to make the process as efficient as possible. Once all the results (or error indications) are received at the Middleware, they are forwarded to the client and the Middleware prepares for the next query to be executed.

Figure 3 describes the components found at the Middleware. There is a *SQL Parser* which takes care of performing a syntactic and semantic check on the query been processed. We assume that SQL is the language used to express all quieres in the system. The SQL Parser generates an internal representation of the query that can be understood and used by the *Query Optimizer*. The optimizer generates an efficient (but not necessarly optimal) plan to execute the query at hand. The criteria used to determine efficiency will be either total time of resource usage, as in [ML86], or response time as proposed in ([GHK92]). The query execution plan is represented as an operator tree, where the leaves represent base relations and the internal nodes represent relational operators such as selection and joins. *Bushy plans* are known to be more efficient than *left-deep* plans for parallel and distributed environments ([IK91], [LVZ93]). Therefore, the optimizer can use either dynamic progamming techniques ([SAC+79]) or randomized algorithms ([IK90]) to generate bushy plans.

The plan produced by the optimizer only contains the join ordering and the placement of selections relative to joins. The sites at which each operation is to be executed are selected at run time. The idea is to wait until knowledge about parameters such as machine load, network bandwith and selectivities become available to help the Middleware in finding the best location the execute each operator.

The *Java Runtime Execution Environment* provides the optimizer with classes that represent the costs of built-in and user-defined operations written in Java. Thus, the optimizer is extensible since we can add new cost functions as needed to compute the costs of the new operations introduced into the system. Sections 5.3 and 5.3.4 give a more detailed description about the use and structure of these Java classes. The job of the *Plan Decomposer* is to find the sources of parallelism in the plan. These sources of parallelism might come from sorting operations, aggregates or the execution of two independent or pipelined operators in the plan. The main goal is to produce enough information to generate a schedule for the parallel execution of the tasks needed to complete the evaluation of the query.

Once the initial plan has been parallelized, it is either stored on disk for future use or given to the *Startup Engine*. At the startup engine, the preparation of the execution plan is finished. Using information obtained from catalogs, past query executions and by contacting some data sources, a picture about current system performance and data location is obtained. This information includes factors such machine load, parameters bound to predicates, available network bandwidth and relation sizes. Based on all these parameters, the location at which each operator in the plan is to be executed is finally determined. The *Operation Coordinator* takes this completed plan and distributes all tasks to the Backend processes running at the proper data sources. The coordinator exchanges control information and results with the backends and the client for the duration of the plan execution.

The final component in the Middleware is the *Network Manager*. This component keeps track of all the network connections used while evaluating a query. It also provides buffers to hold messages and results until the coordinator is ready to process them. The network manager gathers statistics about observed network bandwidth to and from the data sources, which can be used by the optimizer or plan startup engine.

5.2 Backend Level

The *Backend* is the work horse of the system. It contains the necessary machinery to execute the queries users formulate to the Middleware. This machinery includes code to perform selections, joins, compute aggregates and evalute predicates used in the query. Figure 4 shows the basic structure of the Backend.

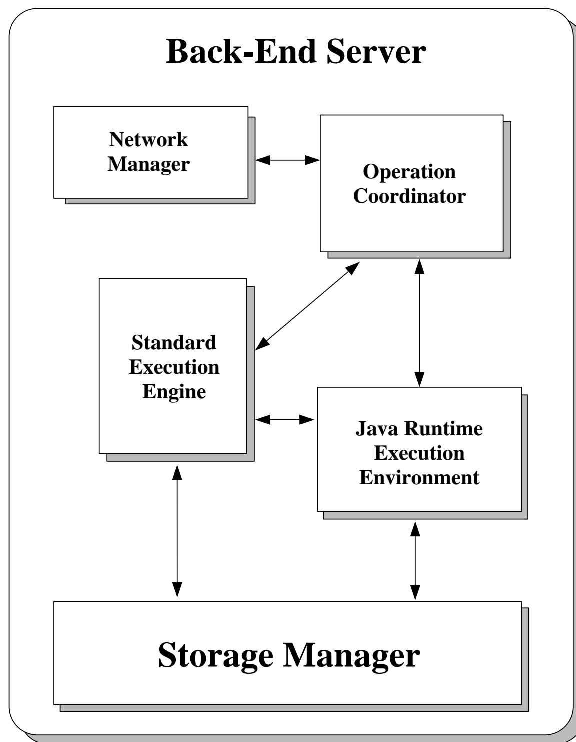

Figure 4: Backend component.

The *Network Manager* has the same functions as in the middleware (see previous section). The *Operation Coordinator* takes care of prossesing control information recieved from the Middleware and/or Backends. It also handles the transfer of all results obtained from operations performed locally.

The *Standard Execution Engine* contains the code the execute the basic operations of the DBMS. This is the typical execution engine found in most DBMS. Here is where relational operators, aggrates, etc., get executed. The functionality in this module is fixed and the user cannot add or remove any part of it. The execution engine has access to the *Storage Manager* and to the *Java Runtime System*. The *Java Runtime Execution Environment* provides the functionality of *mobile code* for relational operators, predicates, etc. Its services can be called from either the *Coordinator* or the *Execution Engine*. The *Java Runtime Execution Environment* contains the necessary class libraries which allow users to write and run their own versions of the operations that the system needs to perform. Section 5.3 gives a more detailed description of the characteristic of this component. Finally, the *Storage Manager* provides services such as buffer management, file scans, index scans, concurrency control and transaction semantics.

We propose that query execution be carried out using an interface similar to that of the iterator model as described in [Gra93]. Under this interface, the classes that are used to implement the iterators for each of the relational operators and index scans contain at least four methods, namely `open`, `produce_next`, `get_next` and `close`. A call to `open` initializes the iterator by allocating memory, opening the necessary files and performing any other operation needed before data can be obtained. In order to generate tuples at the iterator, the `produce_next` method is called repeatedely. To fetch data from an iterator, the `get_next` method is called. Each call to this method will return a block worth of tuples from the iterator.

Notice that we do not associate tuple generation with the `get_next` method as done in [Gra93]. Our main goal with this is to avoid the serialization that such scheme introduces. The local execution engine (the producer) will call the `produce_next` method to generate new tuples. These tuples will then be buffered until another iterator (the consumer) running at a remote execution engine or the middleware invokes the `get_next` method of the local iterator to transfer these tuples in blocks. Obviously, the local site can produce only as many tuples in advance as it can fit in the buffers. Finally, the `close` method is called to deallocate the resources held by the iterator. Some other methods might be needed for housekeeping purposes. For example, there might be a method that indicates the size of the data blocks been returned by the iterator.

5.3 Java Runtime Execution Environment

The Java Runtime Execution Environment (JREE) is the piece of the architecture that enables code mobility between sites. The JREE is shown in figure 5 on page 12. It is an *enhanced* but *compatible* version of the Java Runtime System ([LY97]), [MD97]). Java code shipped in from other places or written by local users is loaded into this module for execution. The code is loaded, verified and either interpreted or translated into the native code of the underlying machine for execution. This code is simply a binary representation of a Java class, which was compiled by the user into Java **bytecode** with a compiler.

The JREE contains an *Application Programming Interface* through which the external Java classes (in bytecode form) implementing the DBMS operations to be performed are introduced into the system. A set of *Base Class Libraries* are used to define the basic functionality of the code that can be run by the JREE. The variables and methods contained in these classes define what the semantics of operations such as joins, index scans and predicate evaluation mean. They serve as a mold, a framework upon which users can define their own operations by simply deriving new classes from these. This is very similar to the approach taken by a Web browser to handle applets. The Web browser contains an applet base class that defines the basic behavior of an applet. Users define new applets by creating new classes which are derived from the applet base class. Then, the basic methods defined by the base class are specialized to behave according to the user needs. When a Web browser downloads and runs applets, it simply calls the well-known set of methods specified by the base class and executes them in the way that the user indicated in her implementation of the derived class.

Clearly, we need to have base classes that indicate what the basic bahavior of relational operators, access methods and predicates is. Users of the DBMS will then use these base

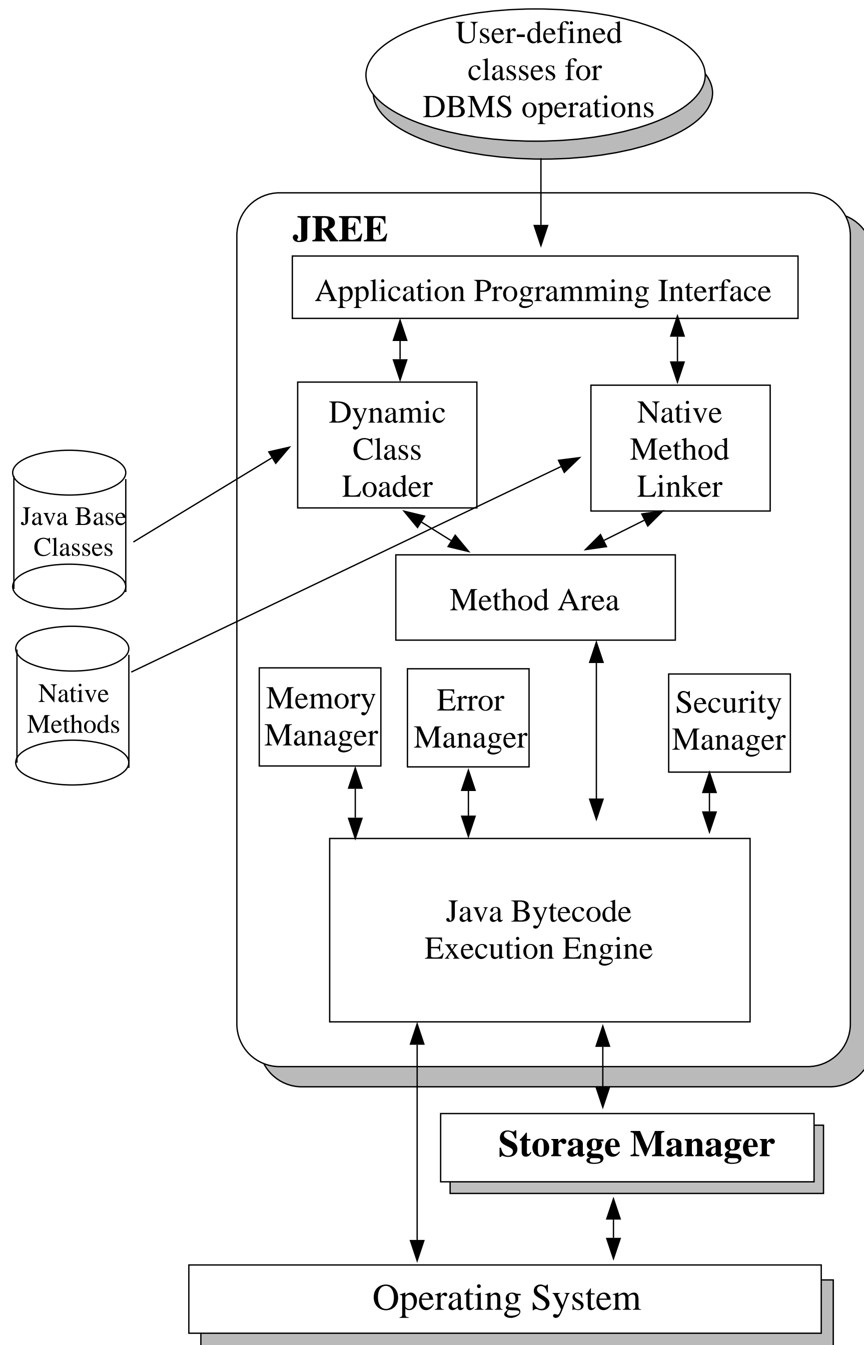

Figure 5: Java Runtime Execution Environment.

classes to implement their own versions of the DBMS operations. These derived classes can then be shipped to a remote DBMS where they are needed because either they are not supported by the system there or their semantics there is different from that at the local DBMS. Figure 6 gives a schematic representation of this idea. In a sense the process of code shipping is analog to the process of downloading and running an applet from a Web server into a Web browser. The DBMS receives or downloads code from another DBMS that can be used to perform a given operation. Thus, we have DBMS functionality moving around the network, helping sites to help themselves.

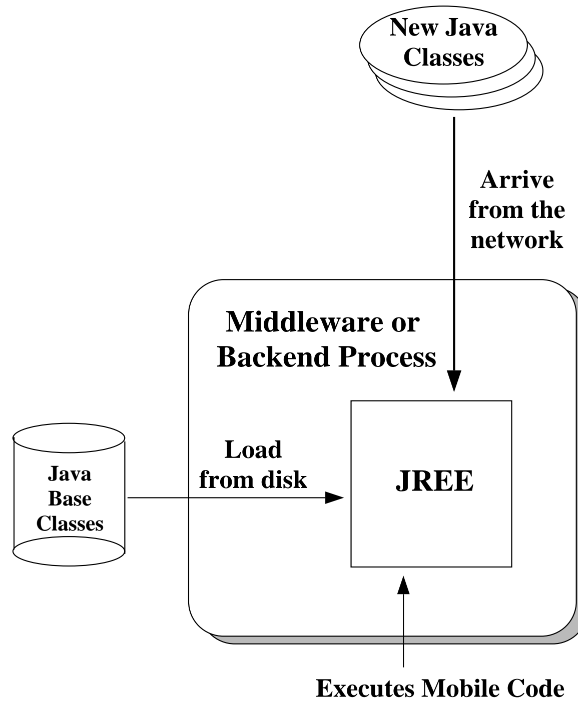

Figure 6: User-defined Java classes been imported into the JREE.

In addition to the base classes that define the behavior of the functions that can be shipped among DBMSs, some other general purpose classes are needed by the JREE. These include classes for I/O, threads, data structures, memory management, etc., which will enable the JREE to complete the tasks at hand. These classes can be written in Java or, in some cases, in some other language such as C or C++. The latter case produces what is known as *native code*. An interface is needed to translate the data types between Java and the language used to create the native code. Sun's *Java Native Methods Interface* provides the fuctionality to perform this translation. The native code is stored in a library that can be linked to the JREE, much like the way shared libraries can be dynamically linked to a UNIX process. The *Dynamic Class Loader* takes care of loading, verifying and storing the bytecode of the Java classes (both base and derived) needed at runtime. The *Native Method Linker* does the job of loading and linking the native code libraries. All loaded classes and their methods are stored in the *Method Area*, from which they can be retrieved and used by the *Java Execution Engine*.

The *Java Execution Engine* contains either a Java bytecode interpreter or a Just-In-

Compiler or both. The bytecode interpreter takes the bytecode from a class and emulates all the operations in software. The Just-In-Time compiler, on the other hand, first converts the bytecode into native machine code and then executes this code. Clearly, the Just-In-Time compiler provides better support, since native machine code will run faster. Although not considered here, there are some hardware implementations of the Java Virtual Machine which could be used to run the bytecode (i.e. Sun's JavaStations). The engine is able to access the underlying operating system and database storage manager by using the support class libraries metioned before.

There is also a *Memory Manager* which manipulates the memory used by the components of the JREE. This includes the memory used by the instances of the classes been executed inside the execution engine. The *Error Manager* handles the exceptions and errors found by the execution engine while executing the code in any class. Exceptions are used to signal the ocurrence of some event that might cause problems. Lost network connections, reading past the end of a file and memory allocation problems are the type of events that will cause an exeception to be raised. The final piece of the JREE is a *Security Manager* which takes care of enforcing proper class behavior. Any type of restriction put into a class is implemented at this module. For example, the Security Manager could enforce the restriction that a file or network connection cannot be opened from within a predicate.

The next four subsections will give details about how the base classes used to define the behavior of mobile code could be structured. The description uses Java-like pseudocode to give specific examples of the possible implementation of these classes.

5.3.1 Predicates

In order to support mobile predicates we ought to have a base class that defines what a predicate does. For example we could have the following:

```
class SQL_Predicate {
    boolean execute(void){
        // execute the predicate code
    }
}
```

Base on this an user can define its own *Equal* predicate for integers by *inherinting* from the `SQL_Predicate` class:

```
class Equal extends SQL_Predicate {
    boolean execute(int  n1, int n2){
        if (n1 == n2)
            return (true)
        else
            return (false)
    }
}
```

In addition to exporting the basic operations that a predicate should perform, the base class might include some type of specification for the kinds of the operations that cannot be

executed inside the predicate. For example, it might specify that no files can be opened at the local file system. This serves for the purpose of enforcing security policies to keep the code from performing unauthorized operations. The specification has to be implemented by the JREE's Security Manager.

5.3.2 Relational Operators

Asumming an interface similar to that of the iterator model ([Gra93]) for the relational operators, we need to have a Java base class interface that indicates the basic three methods in an iterator:

```
interface Iterator {
    int open(void);
    tuple produce_next(void);
    tuple_block get_next(void);
    int close(void);
}
```

From this, the base classes for the relational operator can be implemented. Each operator implements the basic three methods indicated by the `Iterator` interface. For example, a join could be defined as follows:

```
class Join_Iterator implements Iterator{
    final int open(void){
        left_child.open();
        right_child.open();
        . . .
    }

    tuple produce_next(void){
        // put your favorite join code here
    }

    tuple_block get_next(void){
        // put code the ship data in blocks
    }

    final int close(void){
        left_child.close();
        right_child.close();
        . . .
    }
    Iterator left_child, right_child;
}
```

Notice that we have defined the class to have two variables, `left_child` and `right_child`. These two variables simply represent the two sources from which the tuples to be joined

are received. They are defined as iterators, since data comes from some other iterators such as joins or selections. Clearly, this information about that sources must be included in the query execution plan and sent to each execution site. To implement a *hash join* one simply writes a class for it that inherits from the `Join_Iterator` class and specializes the `produce_next` method by adding the necessary code for the hash join. The `final` keyword is used to specify that a method cannot be modified by the users. Again, a more detailed security specification needs to be included to guarantee proper behavior.

5.3.3 Access Methods

The base class for access methods provides basic functionality such as opening and closing of index files. The users can then define the way in which entries are inserted, accessed and deleted from the index files. Indices also follow the iterator interface. A base class for access methods could be written as follows:

```
class Access_Method implements Iterator{
    final int open(String name){
        // use OS interface to open the file
    }

    int insert(key new_key){
        // put insert code here
    }

    tuple produce_next(key search_key){
        // put lookup code here
    }

    tuple_block get_next(void){
        // put code to ship data blocks
    }
    int delete(key search_key) {
        // put delete code here
    }

    final int close(filedescriptor fd){
        // code to close a file
    }

    final int drop_file(String name){
        // code for removing a file from the system
    }
}
```

With this functionality, adding a R-tree index requires the user to write a class derived from `Access_Method` and write the `insert`, `produce_next` and `delete` methods. As indi-

cated before, security must be enforced for the `Access_Method` class by including the proper restriction mechanisms at the Security Manager.

5.3.4 Cost Model Functions

The base class libraries defined before are most likely to be found at the Backend process since they are related to query execution. But since the user is allowed to extend the functionality of the DBMS, it is necessary to inform the query optimizer about the costs of these new functions. Therefore, classes that specify the cost of predicates, relational operators and access methods are needed at the Middleware. One possible implementaion of a base class for cost could be :

```
class Cost_Model {
    int CPU_Cost(void){
    }

    int IO_Cost(void){
    }

    int Network_Cost(void){
    }
}
```

The class `Cost_Model` gives the methods needed to determine the CPU, I/O and Network cost of a particular function. It is necesarry to generate a derived class from `Cost_Model` to tell the optimizer what cost does a user-defined predicate, access method or relational operator has. This process can be done by hand or can be automated by creating the appropriate tools.

5.4 Legacy Data Sources

We now turn to the problem of supporting code mobility at sources where we cannot add a JREE to the existing DBMS or data server. The solution we propose is the use of a modified Backend process that works on top of the existing system running at the data source. This Backend will be an external process, as shown in figure 7, having translation and query execution capabilites. As we mentioned before, it is smarter than a wrapper since it can execute mobile DBMS code.

Since the main goal of the external Backend is to support DBMS mobility, it should be written entirely in Java to make its implementation easier. Figure 8 shows the modifications needed to make the Backend an external process. The *Network Manager* and *Operation Coordinator* have the same tasks as before. The *Execution Engine* executes all operations related to answering a query and has the base classes that support DBMS mobility. Compared with the original Backend (see fig. 4), the duties of the JREE and the Standard Execution Engine have been merged into the Execution Engine of the modified Backend. Notice that before we needed the JREE to make the DBMS understand Java and thus support mobility. Since the external backend is written in Java, the JREE is not needed. All the support base

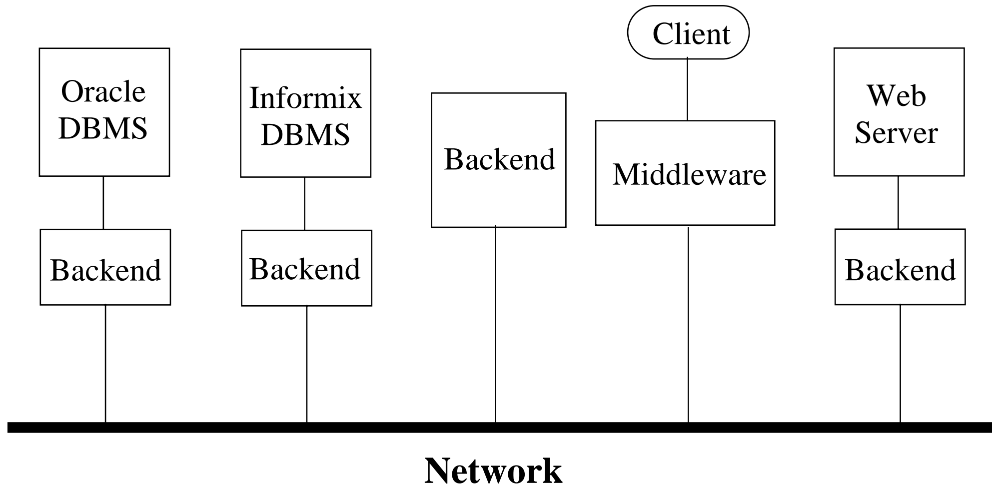

Figure 7: Use of a Backend as an enhanced wrapper.

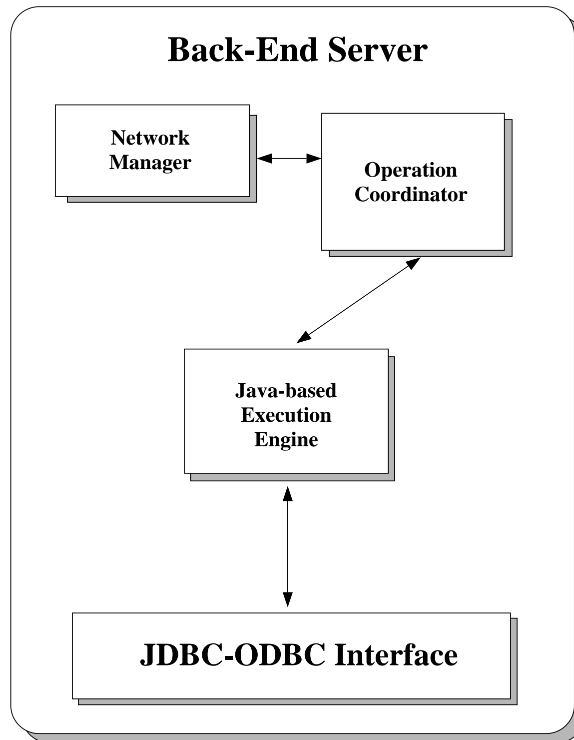

Figure 8: Changes to the Backend to work as an enhanced wrapper.

classes will be provided by the Java Runtime System (i.e. Sun's Java Developers Kit) used to run the external Backend. The *JDBC-ODBC* interface is used by the Backend to access data at the DBMS or server.

Probably, implementing an external Backend will be easier than integrating a JREE into an existing DBMS as discussed in sections 5.1, 5.2 and 5.3. The main drawback of using an external Backend is that we are limited to the operations that can be exported using JDBC or ODBC. With an integrated system, the DBMS internals are accessible and we can have a more efficient (but complex) implementation.

6 Summary

In this report we have proposed a new architecture that supports DBMS mobility. The main idea behind DBMS mobility is the ability to allow parts of a DBMS to migrate to other DBMSs to provide new features at the receiving sites. These parts are simply the code of user-defined predicates, relational operators and indices. By means of this process, the DBMS that gets the new code is dynamically extended. We have presented the basic structure of a Distributed DBMS based on our architecture. The DBMS's functionality is divided into a Middleware process and a Back-End process. Client applications issue queries to and get results from the Middleware. The Middleware performs query processing operations such as parsing and optimization. The Back-End process executes the queries issued by the Middleware on behalf of the clients. Operations such as selections, joins and predicate evaluation occur at the Back-End. Code mobility is realized by writting the code in Java and integrating a Java Runtime Execution Environment at both the Middleware and the Back-End. The Java Runtime Execution Environment executes all the mobile DBMS code. To handle legacy data sources, we have proposed the use of a modified external Back-End as an enhanced wrapper. This modified Back-End, which is to be written in Java, has translation and query execution capabilities. It also supports DBMS mobility.

In order to implement the ideas presented in this report, we could the follow these steps:

- Step 1 - **External Back-End**: This will allow us to create a simple prototype where we can develop the cost models, library classes and algorithms to support code mobility. We can start with predicate mobility, selections and projections. Then joins and index mobility can be added. A fix set of hardwired queries can be used for performance measurements.

- Step 2 - **Middleware**: Once we have the external back-end, we can create a Middleware written in Java and obtain an initial complete system. The cost models developed in step 1 are then integrated into the Middleware, together with algorithms to parallelize the plan. Finally, a more rigorous performance study is conducted.

- Step 3 - **JREE Integration**: The DBMS code of a system such as Postgres or PREDATOR can be used as the base to create a Middleware and a Back-End with an integrated Java Runtime Execution Environment. This can then be used as a reference implementation of our architecture. More performance studies can be conducted here.

References

[Ae76] M. Astrahan and et.al. System R: Relational Approach to Database Management. *ACM Transactions of Database Systems*, 1(2):97–137, 1976.

[CGMH+94] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proc. of IPSJ Conference*, Tokyo, Japan, 1994.

[DR92] Alexios Delis and Nick Roussopoulos. Performance and Scalability of Client-Server Database Architectures. In *Proc. 18th VLDB Conference*, pages 610–623, Vancouver, British Columbia, Canada, 1992.

[FJK96] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *Proc. ACM SIGMOD Conference*, pages 149–160, Montreal, Quebec, Canada, 1996.

[GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query Optimization for Parallel Execution. In *Proc. ACM SIGMOD Conference*, pages 9–18, San Diego, California, USA, 1992.

[GM95] J. Gosling and H. McGilton. The Java Language Environment. Sun Microsystems White Paper, October 1995.

[Gra93] Goetz Grafe. Query Evaluation Techniques for Large Databases. *ACM Computer Surveys*, 25(2):73–170, June 1993.

[IK90] Yannis E. Ioannidis and Younkyung Cha Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proc. ACM SIGMOD Conference*, pages 312–321, Atlantic City, New Jersey, USA, 1990.

[IK91] Yannis E. Ioannidis and Younkyung Cha Kang. Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization. In *Proc. ACM SIGMOD Conference*, pages 168–177, Denver, Colorado, USA, 1991.

[LVZ93] Rosana S.G. Lanzelotte, Patrick Valduriez, and Mohamed Zaït. On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces. In *Proc. 19th VLDB Conference*, pages 493–504, Dublin, Ireland, 1993.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, Massachusetts, 1997.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine.* O'Reilly, Sebastopol, California, 1997.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proc. 12th VLDB Conference*, Kyoto, 1986.

[PP96] Nelson Padua-Pérez. Performance Analysis of Relational Operator Execution in N-Client 1-Server DBMS Architectures. Technical report, University of Maryland, College Park, 1996.

[RCK+95] Nick Roussopoulos, Chungmin M. Chen, Stephen Kelley, Alex Delis, and Yannis Papakonstantinou. The ADMS Project: Views "R" Us. *IEEE Data Engineering Bulletin*, 18(2), 1995.

[RS97] Mary Tork Roth and Peter Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *23rd VLDB Conference*, Athens, Greece, 1997.

[SAC+79] P. Griffiths 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*, pages 23–34, Boston, Massachusetts, USA, 1979.

[SAD+94] Michael Stonebraker, Paul M. Aoki, Robert Devine, Witold Litwin, and Michael Olson. Mariposa: A New Architecture for Distributed Data. In *Proc. 10th Int. Conf. on Data Engineering*, pages 54–65, Houston, Texas, 1994.

[SAP+96] Michael Stonebraker, Paul M. Aoki, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal*, 1996.

[SK91] Michael Stonebraker and Greg Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, 1991.

[SP97] Praveen Seshadri and Mark Paskin. PREDATOR: An OR-DBMS with Enhanced Data Types. In *Proc. ACM SIGMOD Conference*, pages 568–571, Tuczon, Arizona, USA, 1997.

[Sto97] Michael Stonebraker. Architectural Options for Object-Relational DBMSs. Informix White Paper, February 1997.

[SWKH76] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. The Design and Implementation of INGRES. *ACM Transactions on Database System*, 1(3):189–222, 1976.

[TRV96] Anthony Tomasic, Louiqa Rashid, and Patrick Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. In *Proc. Int. Conf. on Distributed Computer Systems*, 1996.

[WDH+81] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, Wilms P, and R. Yost. R*: An Overview of the Architecture. Technical Report RJ3325, IBM Almaden Research Center, San José, California, 1981.