# M A R I P O S A

DISTRIBUTED DATABASE MANAGEMENT SYSTEM

# USER’S MANUAL

# TABLE OF CONTENTS

**1. INTRODUCTION ........ 5**

1.1 WHAT IS MARIPOSA? ........ 5
1.2 USING THIS MANUAL ........ 6
1.3 OVERVIEW OF THE ARCHITECTURE ........ 6

**2. POSTGRES ........ 8**

2.1 THE QUERY LANGUAGE, POSTGRES SQL ........ 9
*2.1.1 Creating a New Class ........ 10*
*2.1.2 Populating a Class with Instances ........ 10*
*2.1.3 Querying a Class ........ 11*
*2.1.4 Redirecting SELECT Queries ........ 12*
*2.1.5 Joins Between Classes ........ 13*
*2.1.6 Updates ........ 13*
*2.1.7 Deletions ........ 14*
*2.1.8 Using Aggregate Functions ........ 14*
2.2 ADVANCED POSTGRES SQL FEATURES ........ 15
*2.2.1 Inheritance ........ 15*
*2.2.2 Time Travel ........ 16*
*2.2.3 Non-Atomic Values: Arrays ........ 16*
2.3 POSTGRES EXTENSIBILITY ........ 18
*2.3.1 The POSTGRES Type System ........ 18*
*2.3.2 About the POSTGRES System Catalogs ........ 18*
2.4 EXTENDING SQL: FUNCTIONS ........ 21
*2.4.1 Query Language (SQL) Functions ........ 21*
2.4.1.1 SQL Functions on Base Types ........ 21
2.4.1.2 SQL Functions on Composite Types ........ 21
*2.4.2 Programming Language Functions ........ 23*
2.4.2.1 Programming Language Functions on Base Types ........ 23
2.4.2.2 Programming Language Functions on Composite Types ........ 25
2.4.2.3 Caveats ........ 26
2.5 EXTENDING SQL: TYPES ........ 27
*2.5.1 Functions Needed for a User-Defined Type ........ 27*
2.6 EXTENDING SQL: OPERATORS ........ 29
2.7 EXTENDING SQL: AGGREGATES ........ 29
2.8 INTERFACING EXTENSIONS TO INDICES ........ 31
2.9 THE POSTGRES RULE SYSTEM ........ 36

**3. MARIPOSA ........ 37**

3.1 MARIPOSA MODULES ........ 37
3.2 A DISTRIBUTED EXAMPLE ........ 38
*3.2.1 Creating a Mariposa Class ........ 39*
*3.2.2 Splitting a Class into Fragments ........ 40*
*3.2.3 Moving Fragments ........ 41*
*3.2.4 Copying a Fragment ........ 42*
3.3 THE MARIPOSA REPLICA SYSTEM ........ 42
*3.3.1 Creating a Copy ........ 43*
*3.3.2 Dropping a Copy ........ 43*
*3.3.3 Moving a Copy ........ 43*
3.4 MARIPOSA NAME SERVICE ........ 43
*3.4.1 Setting Up Name Service ........ 44*

*3.4.2 Specifying A Primary Name Server* ........ *45*
3.5 THE MARIPOSA DATA BROKER ........ 45
3.6 QUERY PROCESSING IN MARIPOSA ........ 47
*3.6.1 The Fragmenter* ........ *47*
3.6.1.1 Fragmented Query Plans ........ 47
*3.6.2 The Query Broker* ........ *49*
3.6.2.1 Bid Curves ........ 50
3.6.2.2 Plan Chunks ........ 51
3.6.2.3 Bid Protocols ........ 52
3.6.2.3.1 The Short Protocol ........ 52
3.6.2.3.2 The Long Protocol ........ 53
*3.6.3 The Bidder* ........ *53*
3.6.3.1 Bidding ........ 53
3.6.3.2 The plan and rtable global variables ........ 54
3.6.3.3 The subcontract Command ........ 56
3.6.3.4 Sample Bidder Script ........ 56

**4. ADMINISTERING POSTGRES AND MARIPOSA ........ 58**

4.1 FREQUENT TASKS ........ 58
*4.1.1 Starting the Site Manager* ........ *58*
*4.1.2 Shutting Down the Postmaster* ........ *59*
*4.1.3 Adding and Removing Users* ........ *59*
*4.1.4 Periodic Upkeep* ........ *59*
*4.1.5 Tuning* ........ *59*
4.2 INFREQUENT TASKS ........ 60
*4.2.1 Cleaning Up After Crashes* ........ *60*
*4.2.2 Moving Database Directories* ........ *62*
*4.2.3 Updating Databases* ........ *63*
4.3 DATABASE SECURITY ........ 63
*4.3.1 Kerberos* ........ *63*
4.4 QUERYING THE SYSTEM CATALOGS ........ 63

# 1. INTRODUCTION

## *1.1 What is Mariposa?*

The Mariposa distributed database management system is an ongoing research project at the University of California at Berkeley. Mariposa addresses fundamental problems in the standard approach to distributed data management. We believe that the underlying assumptions traditionally made while designing distributed data managers do not apply to today's wide-area network (WAN) environments. To date, distributed database management systems have been designed for local-area networks (LAN's) with few servers operating within one administrative domain, such as one company or one department within a company. Furthermore, these systems assume uniformity of all processors and network connections within the system. Data movement in these systems is a very "heavyweight" operation and is performed manually by a database administrator. The explosive growth of distributed computing, illustrated by the World Wide Web, dictates an entirely different set of assumptions.

Mariposa allows DBMS's which are far apart and under different administrative domains to work together to process queries. Furthermore, we have introduced an economic paradigm in which processing sites buy and sell data and query processing services. Not only does this aproach reflect the emerging reality of a commercialized Internet, it has also allowed us to address many of the problems inherent in designing a wide-area distributed DBMS. Mariposa has been designed with the following principles in mind:

- **Scalability to a large number of cooperating sites**. In a WAN environment, there may be a large number of sites. Our goal is to scale to 10,000 servers.

- **Local autonomy**. Each site must have control over its resources. This includes which objects to store and which queries to run. Query and data allocation cannot be done by a central, authoritarian query optimizer.

- **Data mobility**. It should be easy and efficient to change the "home" of an object. Preferably, the object should remain available during movement.

- **No global synchronization**. Updates and schema changes should not force a site to synchronize with all other sites. Otherwise, many common operations will have exceptionally poor response time.

- **Easily configurable policies**. It should be easy for a local database administrator to change the behavior of a Mariposa site. A Mariposa system should respond gracefully to changes in user activity and data access patterns to maintain low response time and high system throughput.

## *1.2 Using This Manual*

This manual is divided into two main parts: Section 2 contains a description of POSTGRES, the single-site database management system distributed as part of Mariposa. Readers who are familiar with POSTGRES may want to skim these sections or skip over them entirely. Section 3 describes the Mariposa system itself. This manual assumes that you have already installed Mariposa successfully on all the sites in your system. For information on how to download and install Mariposa, see the *Installation and Setup Manual*.

## *1.3 Overview of the Architecture*

In Mariposa, all distributed DBMS issues (query optimization, data movement, name service, etc.) are reformulated in microeconomic terms. Implementation of the economic paradigm involves a number of entities and mechanisms. In this section, we describe the architecture and process structure of Mariposa. We begin with an example, pictured in Figure 1.

A company that sells widgets has offices in San Francisco, Chicago, New York and Miami. The company's database includes a table called WIDGETS which contains pricing and inventory information on all the company's widgets. The widgets are warehoused in New York and Miami, so the company keeps half the WIDGETS table in New York and the other half in Miami. In Mariposa, splitting a table is called **fragmentation** and the pieces that make up a table are called **fragments**. In the example, the WIDGETS table is fragmented into WIDGETS1 and WIDGETS2.

If the purchasing manager in the San Francisco office wanted to retrieve all the records from the WIDGETS table, she would enter a query into a frontend application. In SQL (Standard Query Language) she would enter "SELECT * FROM WIDGETS". The site where a query is entered, San Francisco in this case, is the **home site**. The purchasing manager's query is sent from the frontend application to the Mariposa program running on the server in San Francisco. The query is passed through a **parser**, which checks for syntactic correctness and performs type checking; an **optimizer**, which produces a **query plan** that describes the order in which different steps in the plan will be executed; and a **fragmenter**, which changes the plan produced by the optimizer to reflect the data fragmentation. The final result produced by the fragmenter is the **fragmented query plan**. In order to do their work, the parser, optimizer and fragmenter need information about data types, fragment location, etc. This information is maintained by a Mariposa **name server**. In the example, the name server is in the Chicago office.

The fragmented query plan describes the operations that will be performed in order to execute the query, and the order in which they will be carried out. In the example, the purchasing manager's query, "SELECT * FROM WIDGETS" is represented by a query plan which scans the two WIDGETS fragments, WIDGETS1 and WIDGETS2, and merges the result. The fragmented query plan is passed to the **query broker**, whose job it is to decide where each piece of the fragmented query plan will be executed. The query broker uses one of two protocols:

- In the *long protocol*, the query broker contacts the **bidder** module at each potential processing site. The broker waits for responses from the bidders before selecting the best ones. The long protocol is illustrated in Figure 1.

- In the *short protocol*, the query broker uses information collected from the name server to decide which sites will process the query. It does not contact the processing sites.

After the query broker has specified the processing sites, the backend's **coordinator** module takes over. The coordinator notifies the remote sites to begin processing, collects the results, and returns the answer to the client program.

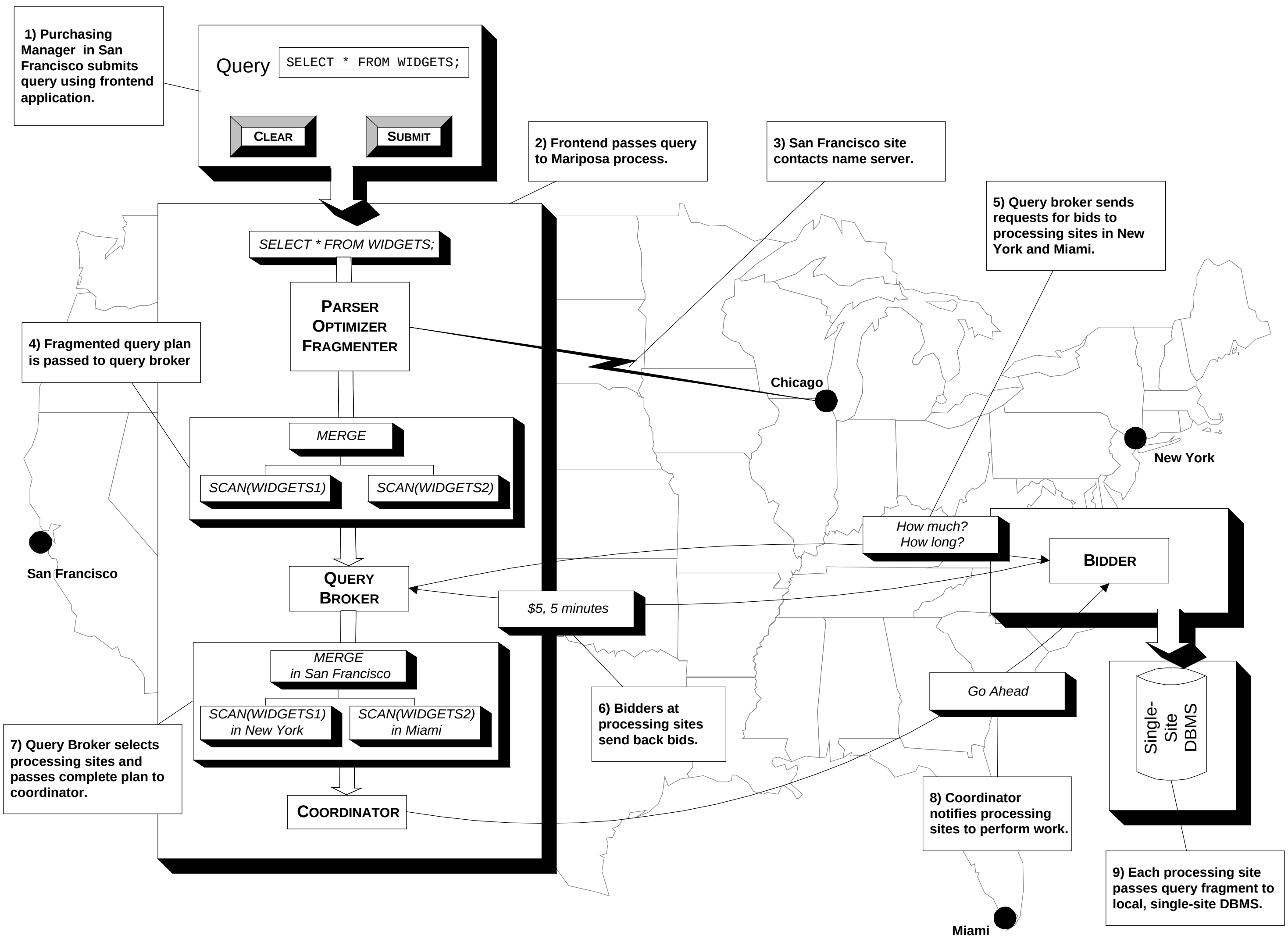



*Figure 1: Mariposa Architecture Example*

This release of Mariposa is based on the POSTGRES extended-relational database management system. We have included a description of the version of POSTGRES distributed with Mariposa in the next section.

# 2. POSTGRES

The single-site database engine distributed with Mariposa is POSTGRES. The version of POSTGRES distributed with Mariposa is a pre-alpha release of POSTGRES95. Not all of the features of POSTGRES95 are implemented in this version. We are planning on releasing a version of Mariposa with support for POSTGRES95 in the future. The commands and keywords listed in Table 1 are not supported by the version of POSTGRES released with Mariposa. We have listed equivalent commands and key words if they exist.

| Command or Key Word | Description | Equivalent |
|---|---|---|
| ASC, DESC | Ascending/Descending key words in ORDER BY clause | USING ‘<‘ for ASC<br>USING ‘>‘ for DESC |
| CAST | Used to typecast constants or parameters | ‘::’ operator |
| COMMIT, ROLLBACK | Transaction commit, rollback | none |
| CREATE DATABASE | Create a new database | CREATEDB |
| DROP DATABASE | Destroy a database | DESTROYDB |
| DELIMITERS | Denotes delimiters between fields in COPY statement | none |
| GRANT, REVOKE, PRIVILEGES, PUBLIC | Used for access control | none |
| EXPLAIN | Explain optimizer choice of query plan | none |
| LIKE | LIKE operator | ‘~’ operator |

*Table 1: Unsupported Commands and Key Words*

In addition to commands and key words, the version of POSTGRES distributed with Mariposa has different built-in types than POSTGRES95. The POSTGRES95 types and their equivalents are listed in Table 2.

| POSTGRES95 Type | Mariposa POSTGRES Type |
|---|---|
| int, integer, smallint | int2, int4 |
| real, float | float4, float8 |
| char(length), varchar(length) | char[length], char16 |
| date, time | abstime |

*Table 2: POSTGRES 95 Types and their Equivalents*

The POSTGRES95 built-in aggregates ***avg***, ***sum***, ***min*** and ***max*** have type-specific equivalents in the POSTGRES distributed with Mariposa. These are listed in Table 3. The built-in aggregate ***count*** is the same in both versions.

| POSTGRES95 Aggregate | POSTGRES Type-Specific Equivalents |
|---|---|
| avg | int4ave, int2ave, float4ave, float8ave |
| sum | int4sum, int2sum, float4sum, float8sum |
| min | int2min, int4min, float4min, float8min |
| max | int2max, int4max, float4max, float8max |

*Table 3: POSTGRES95 Aggregates and their Equivalents*

## *2.1 THE QUERY LANGUAGE, POSTGRES SQL*

This section provides an overview of how to use POSTGRES SQL to perform simple operations. POSTGRES SQL is a variant of SQL-3. It has many extensions such as an extensible type system, inheritance, functions and production rules. These extensions were in the original POSTGRES query language, POSTQUEL.

This manual provides an introduction to POSTGRES SQL. There are numerous books on SQL, such as [MELT93] or [DATE93]; consult them for a more detailed analysis of SQL. Moreover, many of the features of POSTGRES SQL are not part of the ANSI standard.

The following examples assume that you have installed Mariposa on at least one site, and that you have created a database. You must also have the site manager running at the site where you are issuing queries. See the *Installation and Setup Manual*.

The examples in this manual can be found in `src/tutorial.` Refer to the README file in that directory for detailed instructions. To start the tutorial, enter these statements:

```
% cd src/tutorial
% mpsql <database name>
```

The following message will appear:

```
Welcome to the Mariposa interactive sql monitor:
type \? for help on slash commands
type \q to quit
type \g or terminate with semicolon to execute query
You are currently connected to the database: <database-name>

<database-name>=> \i basics.sql
```

The `\i` command reads in queries from the specified files. The `-s` option starts single step mode which pauses before sending a query to the backend. Queries in this section are in the file `basics.sql.`

### 2.1.1 *Creating a New Class*

One of the fundamental concepts in POSTGRES is that of a class. A class is a named collection of object instances. Each instance has the same collection of named attributes of which each attribute is a specific type. Furthermore, each instance has a permanent object identifier (OID) that is unique throughout the installation. Because SQL syntax refers to tables, the terms “table” and “class” are used interchangeably throughout this manual. Similarly, a row is an instance and columns are attributes.

You can create a new class by specifying the class name as well as all attribute names and their types:

```
CREATE TABLE WIDGETS (
   PART_NO   int4,
   LOCATION  char16,   -- warehouse: Miami or New York
   ON_HAND   int4,     -- quantity on-hand
   ON_ORDER  int4,     -- quantity on order
   COMMITTED int4      -- quantity sold but not shipped
);
```

Note that keywords are case-insensitive whereas identifiers are case-sensitive. Therefore, ‘CREATE TABLE’ could have been typed ‘create table’ or ‘Create Table’ but ‘char16’ could not have been typed any other way. POSTGRES SQL supports the standard SQL types (with the exceptions noted in Table 2). POSTGRES is unique in that it can be customized with an arbitrary number of user-defined data types. Consequently, type names are not keywords. For example, you could define a type called ‘CHAR16’ distinct from ‘char16’ and define attributes of type ‘CHAR16’, although this would be confusing, to say the least.

As described so far, the POSTGRES `create table` command is the same command used to create a table in a traditional relational system. However, POSTGRES tables (classes) have properties that are extensions of the relational model.

### 2.1.2 *Populating a Class with Instances*

The insert statement is used to populate a class with instances:

```
INSERT INTO WIDGETS
VALUES (1, ‘New York’, 500, 1500, 300);
```

The copy command is used to load large amounts of data from flat (ASCII) files on the client to the POSTGRES server. For example, the command:

```
COPY WIDGETS FROM ‘src/tutorial/widgets.txt’;
```

will copy the entries in the text file ‘src/tutorial/widgets.txt’ into the WIDGETS table.

### *2.1.3 Querying a Class*

The `WIDGETS` class can be queried with normal relational selection and projection queries. An SQL `select` statement is used to do this. The statement is divided into a target list (i.e., the part that lists the attributes to be returned) and a qualification (i.e., the part that specifies any restrictions). For example, to retrieve all the rows of `WIDGETS`, type:

```
SELECT * FROM WIDGETS;
```

and the output will be:

| PART_NO | LOCATION | ON_HAND | ON_ORDER | COMMITTED |
|---|---|---|---|---|
| 1 | New York | 500 | 1500 | 300 |
| 2 | New York | 3000 | 0 | 1000 |
| 3 | Miami | 10000 | 5000 | 8000 |
| 4 | Miami | 8500 | 0 | 200 |
| 5 | New York | 2500 | 2000 | 2000 |
| 3 | New York | 1800 | 200 | 750 |
| 2 | Miami | 9300 | 700 | 5000 |
| 4 | New York | 3200 | 0 | 0 |
| 6 | New York | 1800 | 5000 | 1500 |
| 6 | Miami | 11000 | 0 | 3000 |

You may specify any aribitrary expressions in the target list. For example, to list the number of widgets on order plus the number on hand, you could type:

```
SELECT PART_NO, LOCATION, (ON_ORDER + ON_HAND) AS TOTAL_QTY
FROM WIDGETS;
```

| PART_NO | LOCATION | TOTAL_QTY |
|---|---|---|
| 1 | New York | 2000 |
| 2 | New York | 3000 |
| 3 | Miami | 15000 |
| 4 | Miami | 8500 |
| 5 | New York | 4500 |
| 3 | New York | 2000 |
| 2 | Miami | 10000 |
| 4 | New York | 3200 |
| 6 | New York | 6800 |
| 6 | Miami | 11000 |

Arbitrary Boolean operators ( e.g., ***and***, ***or***, and ***not***) are allowed in the qualification of any query. For example:

```
SELECT *
FROM WIDGETS
WHERE location = ‘Miami’
and (ON_HAND + ON_ORDER - COMMITTED) <= 8000;
```

| PART_NO | LOCATION | ON_HAND | ON_ORDER | COMMITTED |
|---|---|---|---|---|
| 3 | Miami | 10000 | 5000 | 8000 |
| 2 | Miami | 9300 | 700 | 5000 |
| 6 | Miami | 11000 | 0 | 3000 |

To specify the results of a select to be returned in a sorted order use ORDER BY:

```
SELECT *
FROM WIDGETS
ORDER BY LOCATION;
```

| PART_NO | LOCATION | ON_HAND | ON_ORDER | COMMITTED |
|---|---|---|---|---|
| 6 | Miami | 11000 | 0 | 3000 |
| 2 | Miami | 9300 | 700 | 5000 |
| 4 | Miami | 8500 | 0 | 200 |
| 3 | Miami | 10000 | 5000 | 8000 |
| 2 | New York | 3000 | 0 | 1000 |
| 1 | New York | 500 | 1500 | 300 |
| 3 | New York | 1800 | 200 | 750 |
| 6 | New York | 1800 | 5000 | 1500 |
| 5 | New York | 2500 | 2000 | 2000 |
| 4 | New York | 3200 | 0 | 0 |

To group records together, use GROUP BY. GROUP BY is generally used with aggregates:

```
SELECT PART_NO,
int4sum(ON_HAND) as TOTAL_ON_HAND,
int4sum(ON_ORDER) as TOTAL_ON_ORDER,
int4sum(COMMITTED) as TOTAL_COMMITTED
FROM WIDGETS
GROUP BY PART_NO
ORDER BY PART_NO;
```

| PART_NO | TOTAL_ON_HAND | TOTAL_ON_ORDER | TOTAL_COMMITTED |
|---|---|---|---|
| 1 | 500 | 1500 | 300 |
| 2 | 12300 | 700 | 6000 |
| 3 | 11800 | 5200 | 8750 |
| 4 | 11700 | 0 | 200 |
| 5 | 2500 | 2000 | 2000 |
| 6 | 12800 | 5000 | 4500 |

To find out more about aggregates, see Section 2.1.8.

### *2.1.4 Redirecting SELECT Queries*

Any select query can be redirected to a new class

```
SELECT * INTO TABLE temp from WIDGETS;
```

This query implicitly creates a new class `temp` with the attribute names and types specified in the target list of the SELECT INTO command. Thus, you can perform operations on the resulting class as well as on other classes:

```
SELECT * FROM temp;
```

| PART_NO | LOCATION | ON_HAND | ON_ORDER | COMMITTED |
|---|---|---|---|---|
| 1 | New York | 500 | 1500 | 300 |
| 2 | New York | 3000 | 0 | 1000 |
| 3 | Miami | 10000 | 5000 | 8000 |
| 4 | Miami | 8500 | 0 | 200 |
| 5 | New York | 2500 | 2000 | 2000 |
| 3 | New York | 1800 | 200 | 750 |
| 2 | Miami | 9300 | 700 | 5000 |
| 4 | New York | 3200 | 0 | 0 |
| 6 | New York | 1800 | 5000 | 1500 |
| 6 | Miami | 11000 | 0 | 3000 |

### *2.1.5 Joins Between Classes*

So far this section has shown queries that access one class at a time. Queries can access multiple classes at once, or access the same class in such a way that multiple instances of the class are being processed at the same time. A query that accesses multiple instances of the same or different classes at one time is called a *join* query.

For example, to find the widgets are on-hand in greater quantity in Miami than in New York:

```
SELECT W1.PART_NO, W1.ON_HAND as MIAMI, W2.ON_HAND as NY
FROM WIDGETS W1, WIDGETS W2
WHERE W1.LOCATION = ‘Miami’ and
W2.LOCATION = ‘New York’ and
W1.PART_NO = W2.PART_NO and
W1.ON_HAND > W2.ON_HAND;
```

| PART_NO | MIAMI | NY |
|---|---|---|
| 2 | 9300 | 3000 |
| 3 | 10000 | 1800 |
| 4 | 8500 | 3200 |
| 6 | 1100 | 1800 |

In this case, both `W1` and `W2` are surrogates for an instance of the class `widgets,` and both range over all instances of the class. In relational database systems, `W1` and `W2` are known as “range variables.” In addition, a query can contain an arbitrary number of class names and surrogates.[1]

### *2.1.6 Updates*

To update existing instances, use the `update` command. For example, to reflect the delivery into New York of the widgets with PART_NO = 1 that were on order:

```
UPDATE WIDGETS
```

[1] The semantics of such a join are that the qualification is a truth expression defined for the Cartesian product of the classes indicated in the query. For those instances in the Cartesian product for which the qualification is true, POSTGRES computes and returns the values specified in the target list. POSTGRES SQL does not assign any meaning to duplicate values in such expressions. This means that POSTGRES sometimes recomputes the same target list several times—this frequently happens when Boolean expressions are connected with an or. To remove such duplicates, you must use the SELECT DISTINCT statement.

```
SET ON_HAND = ON_HAND + ON_ORDER, ON_ORDER = 0
WHERE PART_NO = 1 and
LOCATION = 'New York';

SELECT * FROM WIDGETS
WHERE PART_NO = 1 and
LOCATION = 'New York';
```

| PART_NO | LOCATION | ON_HAND | ON_ORDER | COMMITTED |
|---|---|---|---|---|
| 1 | New York | 2000 | 0 | 300 |

### *2.1.7 Deletions*

Deletions are performed using the delete command:

```
DELETE FROM WIDGETS
WHERE ON_HAND = 0 and
ON_ORDER = 0 and
COMMITTED = 0;
```

All widgets with zero quantity on-hand, on-order and sold are deleted. (In this example, there are no such records, so this DELETE statement has no effect).

Be wary of queries such as:

```
DELETE FROM WIDGETS;
```

Without a qualification, the `delete` command deletes all instances of the given class, leaving it empty. The system will not request confirmation before performing this command.

### *2.1.8 Using Aggregate Functions*

As in most other query languages, POSTGRES supports aggregate functions. However, in the current implementation of POSTGRES, the usage of aggregate functions is limited. Specifically, while there are aggregates to compute such functions as the count, sum, average, maximum and minimum over a set of instances, aggregates can only appear in the target list of a query and not in the qualification ( i.e, the `where` clause). For example:

```
SELECT int4max(ON_HAND) as MAX_ON_HAND
FROM WIDGETS;
```

| MAX_ON_HAND |
|---|
| 11000 |

However, this query won't be accepted by POSTGRES:

```
SELECT PART_NO, LOCATION, ON_HAND
FROM WIDGETS
WHERE ON_HAND = int4max(ON_HAND);
```

As mentioned in Section 2.1.3, aggregates are commonly used with GROUP BY clauses.

## 2.2 ADVANCED POSTGRES SQL FEATURES

This section discusses those features that distinguish POSTGRES from conventional data managers. These features include inheritance, time travel, and non-atomic data values (i.e., array- and set-valued attributes).

Examples in this section can be found in `advance.sql` in the tutorial directory. (Refer to the introduction of the previous chapter for details).

### *2.2.1 Inheritance*

The following examples illustrate inheritance in POSTGRES. The statements below create the class cities as well as the capitals class which contains all the state capitals. The `capitals` class inherits from the `cities` class.

```
CREATE TABLE cities (
name      text,
population   float8,
altitude     int4 -- (in ft)
);

CREATE TABLE capitals (
state char2
) INHERITS (cities);
```

In this case, an instance of `capitals` inherits all attributes (i.e., `name, population,` and `altitude)` from its parent, `cities.` The type of the attribute `name` is `text,` a built-in POSTGRES type for variable-length strings. The type of the attribute `population` is float8, the POSTGRES built-in type for double-precision floating point number. The type for the `altitude` attribute is `int4,` a built-in POSTGRES type for regular four-byte integer numbers.

The `capitals` class has an extra attribute, `state,` which contains the capital's state. In POSTGRES, a class can inherit from zero or more other classes.[2] In addition, a query can reference either all instances of a class or all instances of a class plus all of its descendants. For example, the following query finds all the cities that are situated at an attitude of 500 feet or higher:

```
SELECT name, altitude
FROM cities
WHERE altitude > 500;
```

| name | altitude |
|---|---|
| Las Vegas | 2174 |
| Mariposa | 1953 |

On the other hand, to find the names of all cities, including state capitals, that are located at an altitude over 500 feet, the query is:

```
SELECT c.name, c.altitude
FROM cities* c
WHERE c.altitude > 500;
```

[2] I.e., the inheritance hierarchy is a directed acyclic graph.

which returns:

| name | altitude |
|---|---|
| Las Vegas | 2174 |
| Mariposa | 1953 |
| Madison | 845 |

Here the “*” after `cities` indicates that the query should be run over `cities` and all classes below `cities` in the inheritance hierarchy. Many of the commands discussed so far—`select, update` and `delete`—support this “*” notation. Other commands, such as the `alter` command do as well.

### *2.2.2 Time Travel*

POSTGRES supports time travel. This feature enables you to run historical queries. For example, to find the current population of Mariposa city:

```
SELECT * FROM cities WHERE name = ‘Mariposa’;
```

| name | population | altitude |
|---|---|---|
| Mariposa | 1320 | 1953 |

POSTGRES automatically finds the version of Mariposa’s record valid at the current time.

You can also specify a time range. For example, to retrieve the past and present populations of Mariposa, query:

```
SELECT name, population
FROM cities[epoch’, ‘now’]
WHERE name = ‘Mariposa’;
```

Here, “epoch” indicates the beginning of the system clock.[3] If all of the examples have been executed thus far, then the above query returns:

| name | population |
|---|---|
| Mariposa | 1200 |
| Mariposa | 1320 |

The default beginning of a time range is the earliest time representable by the system and the default end is the current time; thus, the above time range can be abbreviated as ``[,].

### *2.2.3 Non-Atomic Values: Arrays*

One of the tenets of the relational model is that the attributes of a relation are atomic. POSTGRES does not have this restriction; attributes can contain subvalues that can be accessed from the query language. For example, you can create attributes that are arrays of base types.

With POSTGRES attributes of an instance can be defined as fixed-length or variable-length multi-dimensional arrays. Arrays of any base type or user-defined type can be created. To illustrate this, first create a class with arrays of base types.

[3] On UNIX systems, this is always midnight, January 1, 1970 GMT.

```
CREATE TABLE SAL_EMP (
name      text,
pay_by_quarter   int4[],
schedule     char16[][]
);
```

The above query creates a class named `SAL_EMP` with a `text` string (`name`), a one-dimensional array of `int4` (`pay_by_quarter`), which represents the employee's salary by quarter, and a two-dimensional array of `char16` (`schedule`), which represents the employee's weekly schedule.

To insert values into an array, use the `INSERT` statement. Note that when appending to an array, enclose the values within braces and separate them by commas. This is not unlike the syntax for initializing structures in C.

```
INSERT INTO SAL_EMP
VALUES (‘Bill’,
‘{10000, 10000, 10000, 10000}’,
‘{{”meeting”, “lunch”}, {}}’);

INSERT INTO SAL_EMP
VALUES (‘Carol’,
‘{20000, 25000, 25000, 25000}’,
‘{{”talk”, “consult”}, {”meeting”}}’);
```

By default, POSTGRES uses the “one-based” numbering convention for arrays—that is, an array of ***n*** elements starts with array[1] and ends with array[ ***n***].

The following query accesses a single element of an array at a time and retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name
FROM SAL_EMP
WHERE SAL_EMP.pay_by_quarter[1] <>
SAL_EMP.pay_by_quarter[2];
```

| name |
|---|
| Carol |

The following query retrieves the third quarter pay of all employees:

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;
```

You can also access arbitrary slices of an array, (subarrays). The following query retrieves the first item on Bill's schedule for the first two days of the week.

| pay_by_quarter |
|---|
| 10000 |
| 25000 |

```
SELECT SAL_EMP.schedule[1:2][1:1]
FROM SAL_EMP
WHERE SAL_EMP.name = ‘Bill’;
```

| schedule |
|---|
| {{”meeting”},{””}} |

## 2.3 POSTGRES Extensibility

This section discusses how to extend the POSTGRES SQL query language by adding functions, types, operators, and aggregates.

Standard relational systems store information about databases, tables and columns in what are commonly known as system catalogs. (Some systems call this the "data dictionary.") Although the DBMS stores its internal bookkeeping within the system catalogs, this information is typically not available to users.

One key difference between POSTGRES and standard relational systems is that POSTGRES stores much more information in its catalogs than relational systems do—not only information about tables and columns, but also information about its types, functions, access methods, and so forth. These classes can be modified and extended by the user, thereby extending the built-in capabilities of POSTGRES. By comparison, conventional database systems can only be extended by changing hardcoded procedures within the DBMS or by loading modules specially-written by the DBMS vendor.

POSTGRES is also unlike most other data managers in that the server can incorporate code written by users through dynamic loading. That is, a user can specify an object code file (e.g., a compiled `.o` file or shared library) which implements a new type or function and POSTGRES will load it as required. Code written in SQL is even easier to add to the server.

### 2.3.1 The POSTGRES Type System

POSTGRES types are divided into two categories: base types and composite types. Base types are those like `int4`, which are implemented in a programming language such as C. They generally correspond to what are often known as "abstract data types". POSTGRES can only operate on these types through methods provided by the user. Furthermore, POSTGRES understands the behavior of such types only to the extent that the user describes them.

Composite types are created whenever a user creates a class. `WIDGETS` is an example of a composite type. POSTGRES stores all instances of these types in a file. Information about the attributes of composite types are stored in one of the POSTGRES system catalogs (`pg_attribute`) and can by queried like any other table.

POSTGRES base types are further divided into built-in types and user-defined types. Built-in types (like `int4`)are those that are compiled into the system and distributed along with the source code. User-defined types, as the names suggests, are defined by the user. These are described in detail in Section 2.5.

### 2.3.2 About the POSTGRES System Catalogs

All system catalogs have names that begin with "pg_". The following classes contain information that may be useful to the end user. There are other system catalogs, but there should rarely be a reason to query them directly.

| ***catalog name*** | ***description*** |
|---|---|
| pg_database | databases |
| pg_class | classes |
| pg_attribute | class attributes |
| pg_index | secondary indices |
| pg_proc | procedures (both C and SQL) |
| pg_type | types (both base and complex) |
| pg_operator | operators |
| pg_aggregate | aggregates and aggregate functions |
| pg_am | access methods |
| pg_amop | access method operators |
| pg_amproc | access method support functions |
| pg_opclass | access method operator classes |

The POSTGRES Reference Manual gives a more detailed explanation of these catalogs and their attributes. However, Figure 3 shows the major entities and their relationships in the system catalogs. (Attributes that do not refer to other entities are not shown unless they are part of a primary key.)

This diagram becomes clear when you examine the catalogs' contents and see how they relate to each other. The main points are:

- Several of the following sections present various join queries on the system catalogs that display information needed to extend the system. This diagram should make these join queries (which are often three- or four-way joins) more understandable, because the diagram shows that the attributes used in the queries form foreign keys in other classes.

- Many different features (i.e, classes, attributes, functions, types, access methods, etc.) are tightly integrated in this schema. A simple create command may modify many of these catalogs.

- Types and procedures[4] are central to the schema. Nearly every catalog contains some reference to instances in one or both of these classes. For example, POSTGRES frequently uses type signatures (e.g., of functions and operators) to identify unique instances of other catalogs.

- There are many attributes and relationships that have obvious meanings, but there are many that do not; for example, those that have to do with access methods. The relationships between `pg_am`, `pg_amop`, `pg_amproc`, `pg_operator` and `pg_opclass` are particularly hard to understand are in

[4] This manual uses the words procedure and function more or less interchangably.

depth (in the section on interfacing types and operators to indices) following the discussion of basic extensions.

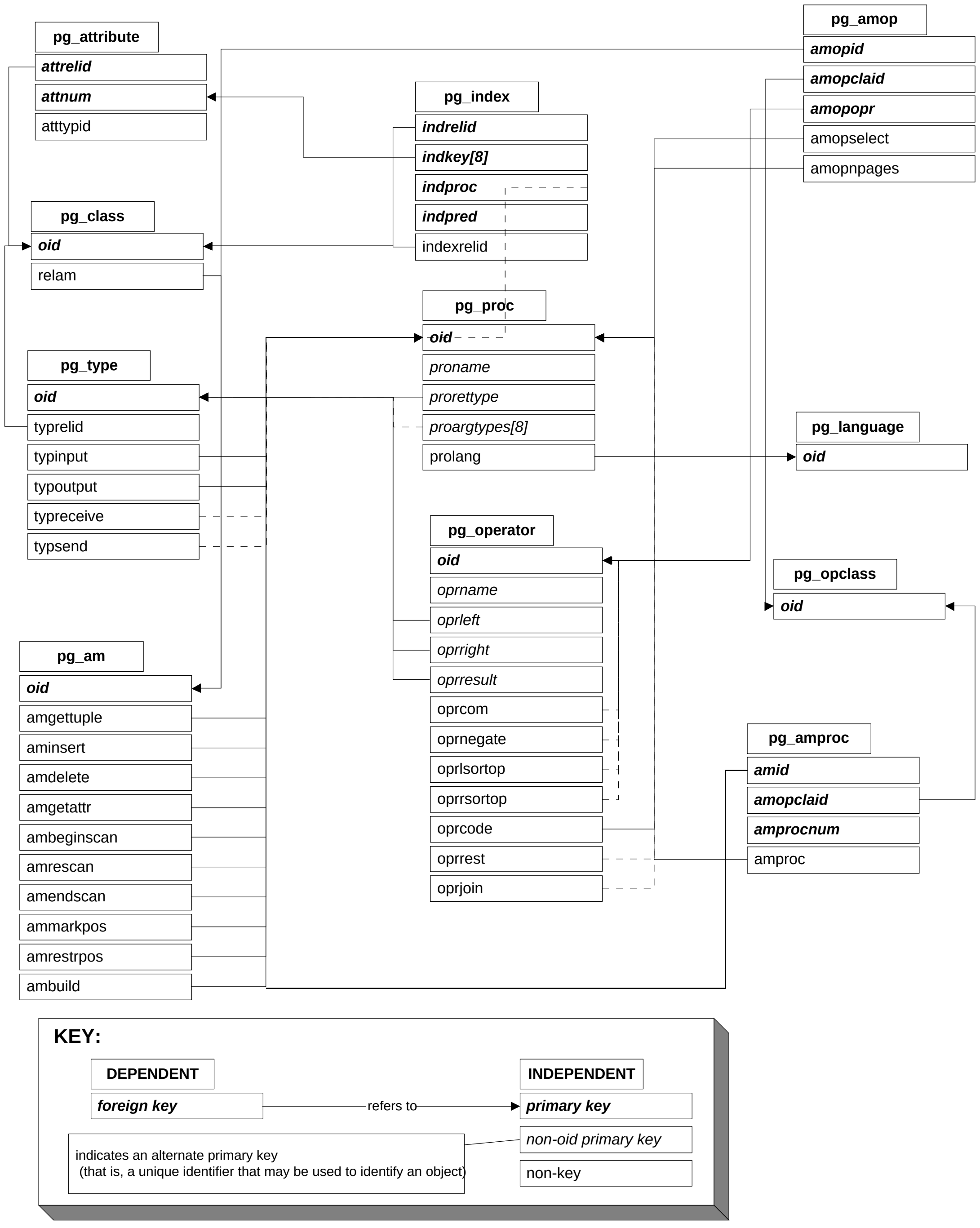



*Figure 2: The major POSTGRES system catalogs.*

## *2.4 EXTENDING SQL: FUNCTIONS*

An important part of defining a new type is the definition of functions that describe the type's behavior. While it is possible to define a new function without defining a new type, the reverse is not true.

POSTGRES SQL provides two types of functions: query language functions (functions written in SQL) and programming language functions (functions written in a compiled programming language, such as C). Both query language functions and programming language functions can take any type of variable as arguments and return any type. This includes base types, composite types or a combination of both.

Examples in this section can be found in `funcs.sql` and `C-code/funcs.c.`

### *2.4.1 Query Language (SQL) Functions*

Query language functions can be input by a POSTGRES user from the command line and stored in the database. They require no programming experience aside from SQL.

#### 2.4.1.1 SQL Functions on Base Types

The simplest possible SQL function has no arguments and simply returns a base type, such as `int4:`

```
CREATE FUNCTION one() RETURNS int4
AS ‘SELECT 1 as RESULT’ LANGUAGE ‘sql’;

SELECT one() AS answer;
```

| answer |
|---|
| 1 |

Notice that the function definition included a target list with the name `RESULT,` but the target list of the query that invoked the function overrode the function's target list. Therefore, the result is labelled `answer` instead of `one.`

It's almost as easy to define SQL functions that take base types as arguments. In the example below, notice how arguments within the function are referred to as "`$1`" and "`$2`".

```
CREATE FUNCTION add_em(int4, int4) RETURNS int4
AS ‘SELECT $1 + $2;’ LANGUAGE ‘sql’;

SELECT add_em(1, 2) AS answer;
```

| answer |
|---|
| 3 |

#### 2.4.1.2 SQL Functions on Composite Types

When specifying functions with arguments of composite types (such as `EMP),` you must not only state which argument you want (as you did above with "`$1`" and "`$2`") but also the attributes of that argument. For example, take the function `double_salary` that computes what your salary would be if it were doubled.

```
CREATE FUNCTION double_salary(EMP) RETURNS int4
AS ‘SELECT $1.salary * 2 AS salary;’ LANGUAGE ‘sql’;

SELECT name, double_salary(EMP) AS dream
FROM EMP
WHERE EMP.dept = ‘toy’;
```

| name | dream |
|---|---|
| Sam | 2400 |

Notice the use of the syntax “`$1.salary`”.

Usually you can use the notation `attribute(class)` and `class.attribute` interchangeably.

- `this is the same as:`
- `SELECT EMP.name AS youngster FROM EMP WHERE EMP.age < 30`

```
SELECT name(EMP) AS youngster
FROM EMP
WHERE age(EMP) < 30;
```

| youngster |
|---|
| Sam |

However, this is not always the case.

Function notation is important when you want to create a function that returns a single instance of a complex type. You do this by assembling the entire instance within the function, attribute by attribute. This is an example of a function that returns a single `EMP` instance:

```
CREATE FUNCTION new_emp() RETURNS EMP
AS ‘SELECT \’None\’::text AS name,
1000 AS salary,
25 AS age,
\’none\’::char16 AS dept;’
LANGUAGE ‘sql’;
```

In this case you have specified each of the attributes with a constant value, but any computation or expression could have been substituted for these constants.

Defining a function like this can be tricky. Some of the more important considerations are:

- The target list order must be exactly the same as that in which the attributes appear in the CREATE TABLE statement (or when you execute a `.*` query).
- You must typecast the expressions (using `::`) very carefully or you will see the following error:

```
WARN::function declared to return type EMP does not retrieve
(EMP.*)
```

- When calling a function that returns an instance, you cannot retrieve the entire instance. You must either project an attribute out of the instance or pass the entire instance into another function:

```
SELECT name(new_emp()) AS nobody;
```

| nobody |
|---|
| None |

Because the parser doesn't understand the other (dot) syntax for projection when combined with function calls, you should use the function syntax for projecting attributes of function return values.

```
SELECT new_emp().name AS nobody;
WARN:parser: syntax error at or near "."
```

Any collection of commands in the SQL query language can be packaged together and defined as a function. The commands can include updates (i.e., insert, update and delete) as well as select queries. However, the final command must be a select that returns whatever is specified as the function's `return type.`

```
CREATE FUNCTION clean_EMP () RETURNS int4
AS 'DELETE FROM EMP WHERE EMP.salary <= 0;
SELECT 1 AS ignore_this'
LANGUAGE 'sql';

SELECT clean_EMP();
```

| ignore_this |
|---|
| 1 |

### *2.4.2 Programming Language Functions*

This section describes programming language functions on base types and on composites.

#### 2.4.2.1 Programming Language Functions on Base Types

Internally, POSTGRES regards a base type as a "blob of memory." User-defined functions over a user-defined base type define the way that POSTGRES operates on the type. That is, POSTGRES will only store and retrieve the data from disk. It will use the user-defined functions to input, process, and output the data.

Base types can have one of three internal formats:

- pass by value, fixed-length
- pass by reference, fixed-length
- pass by reference, variable-length

By-value types can only be 1, 2 or 4 bytes in length (even if your computer supports by-value types of other sizes). POSTGRES itself only passes integer types by value. You should be careful to define your types so that they will be the same size (in bytes) on all architectures. For example, the `long` type is dangerous because it is 4 bytes on some machines and 8 bytes on others, whereas the `int` type is 4 bytes on most UNIX machines (though not on most personal computers). A reasonable implementation of the `int4` type on UNIX machines might be:

```
/* 4-byte integer, passed by value */
typedef int int4;
```

On the other hand, fixed-length types of any size may be passed by-reference. For example, here is a sample implementation of the POSTGRES `char16` type:

```
/* 16-byte structure, passed by reference */
typedef struct {
char data[16];
} char16;
```

Only pointers to such types can be used when passing them in and out of POSTGRES functions.

Finally, all variable-length types must also be passed by reference. All variable-length types must begin with a length field of exactly 4 bytes, and all data to be stored within that type must be located in the memory immediately following that length field. The length field is the total length of the structure (i.e., it includes the size of the length field itself). You can define the `text` type as follows:

```
typedef struct {
int4 length;
char data[1];
} text;
```

Obviously, the `data` field is not long enough to hold all possible strings—it's impossible to declare such a structure in C. When manipulating variable-length types, be careful to allocate the correct amount of memory and initialize the length field. For example, if you want to store 40 bytes in a `text` structure, you might use a code fragment like this:

```
#include “postgres.h”
#include “utils/palloc.h”

...

char buffer[40]; /* our source data */
...

text *destination = (text *) palloc(VARHDRSZ + 40);
destination->length = VARHDRSZ + 40;
memmove(destination->data, buffer, 40);
...
```

Here are some examples of real functions. Suppose `funcs.c` look like:

```
#include <string.h>
#include “postgres.h”  /* for char16, etc. */
#include “utils/palloc.h” /* for palloc */
int
add_one(int arg)
{
    return(arg + 1);
}

char16 *
concat16(char16 *arg1, char16 *arg2)
{
    char16 *new_c16 = (char16 *) palloc(sizeof(char16));
    memset((void *) new_c16, 0, sizeof(char16));
    (void) strncpy(new_c16, arg1, 16);
    return (char16 *)(strncat(new_c16, arg2, 16));
}
```

```
text *
copytext(text *t)
{
    /*
    * VARSIZE is the total size of the struct in bytes.
    */
    text *new_t = (text *) palloc(VARSIZE(t));
    memset(new_t, 0, VARSIZE(t));
    VARSIZE(new_t) = VARSIZE(t);
    /*
    * VARDATA is a pointer to the data region of the  struct.
    */
    memcpy((void *) VARDATA(new_t), /* destination */
    (void *) VARDATA(t), /* source */
    VARSIZE(t)-VARHDRSZ); /* how many bytes */
    return(new_t);
}
```

On OSF/1 you type:

```
CREATE FUNCTION add_one(int4) RETURNS int4
AS   '/usr/local/postgres95/tutorial/obj/funcs.so'   LANGUAGE
'c';
CREATE FUNCTION concat16(char16, char16) RETURNS char16
AS   '/usr/local/postgres95/tutorial/obj/funcs.so'   LANGUAGE
'c';
CREATE FUNCTION copytext(text) RETURNS text
AS   '/usr/local/postgres95/tutorial/obj/funcs.so'   LANGUAGE
'c';
```

On other systems, you might have to make the filename end in `.sl` to indicate that it's a shared library.

#### 2.4.2.2 Programming Language Functions on Composite Types

Composite types do not have a fixed layout like C structures. Instances of a composite type may contain null fields. Also, composite types that are part of an inheritance hierarchy may have different fields than other members of the same inheritance hierarchy. Therefore, POSTGRES provides a procedural interface for accessing fields of composite types from C.

As POSTGRES processes a set of instances, each instance will be passed into your function as an opaque structure of type TUPLE.

Suppose you want to write a function to answer the query

```
SELECT name, c_overpaid(EMP, 1500) AS overpaid
FROM EMP
WHERE name = 'Bill' or name = 'Sam'
```

In the query above, you can define `c_overpaid` as:

```
#include "postgres.h"  /* for char16, etc. */
#include "libpq-fe.h" /* for TUPLE */
bool
c_overpaid(TUPLE t, /* the current instance of EMP */
int4 limit)
{
    bool isnull = false;
    int4 salary;
```

```
        salary = (int4) GetAttributeByName(t, "salary", &isnull);
        if (isnull)
            return (false);
        return(salary > limit);
    }
```

`GetAttributeByName` is the POSTGRES system function that returns attributes out of the current instance. It has three arguments: the argument of type TUPLE passed into the function, the name of the desired attribute, and a return parameter that describes whether the attribute is null. `GetAttributeByName` will align data properly so you can cast its return value to the desired type. For example, if you have an attribute `name` which is of the type `char16,` the `GetAttributeByName` call would look like:

```
    char *str;
    ...
    str = (char *) GetAttributeByName(t, "name", &isnull)
```

The following query lets POSTGRES know about the `c_overpaid` function:

```
    CREATE FUNCTION c_overpaid(EMP, int4) RETURNS bool
    AS   '/usr/local/postgres95/tutorial/obj/funcs.so'   LANGUAGE
    'c';
```

While there are ways to construct new instances or modify existing instances from within a C function, these are far too complex to discuss in this manual.

#### 2.4.2.3 Caveats

This section discusses the more difficult task of writing programming language functions. Be warned: this section of the manual will not make you a programmer. You must have a good understanding of C (including the use of pointers and the `malloc` memory manager) before trying to write C functions for use with POSTGRES.

While it may be possible to load functions written in languages other than C into POSTGRES, this is often difficult (when it is possible at all) because other languages, such as FORTRAN and Pascal often do not follow the same calling convention as C. That is, other languages do not pass argument and return values between functions in the same way. For this reason, this discussion assumes that your programming language functions are written in C.

The basic rules for building C functions are as follows:

- Most of the header (include) files for POSTGRES should already be installed in `/usr/local/mariposa/include`. You should always include `I/usr/local/mariposa/include` on your `cc` command lines. Sometimes, you may find that you require header files that are in the server source itself. In those cases you may need to add one or more of

  ```
  I/usr/local/postgres95/src/backend
  I/usr/local/postgres95/src/backend/include
  I/usr/local/postgres95/src/backend/port/<PORTNAME>
  I/usr/local/postgres95/src/backend/obj
  ```

  (where `<PORTNAME>` is the name of the port, e.g., `alpha` or `sparc).`

- When allocating memory, use the POSTGRES routines palloc and pfree instead of the corresponding C library routines malloc and free. The memory allocated by palloc will be freed automatically at the end of each transaction, preventing memory leaks.

- Always zero the bytes of your structures using memset or bzero. Several routines (such as the hash access method, hash join and the sort algorithm) compute functions of the raw bits contained in your structure. Even if you initialize all fields of your structure, there may be several bytes of alignment padding (holes in the structure) that may contain garbage values.

- Most of the internal POSTGRES types are declared in postgres.h, so it's usually a good idea to include that file as well.

- Compiling and loading your object code so that it can be dynamically loaded into POSTGRES always requires special flags. See Appendix A for a detailed explanation of how to do it for your particular operating system.

## *2.5 EXTENDING SQL: TYPES*

As previously mentioned, there are two kinds of types in POSTGRES: base types (defined in a programming language) and composite types (instances).

Examples in this section up to interfacing indices can be found in `complex.sql` and `complex.c.` Composite examples are in `funcs.sql.`

### *2.5.1 Functions Needed for a User-Defined Type*

A user-defined type must always have input and output functions. These functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in memory. The input function takes a null-delimited character string as its input and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type and returns a null-delimited character string.

Suppose you want to define a `complex` type which represents complex numbers. Naturally, you can represent a complex in memory as the following C structure:

```
typedef struct Complex {
double x;
double y;
} Complex; and a string of the form "(x,y)" as the external
string representation.
```

These functions are usually not hard to write, especially the output function. However, there are a number of points to remember.

- When defining your external (string) representation, remember that you must eventually write a complete and robust parser for that representation as your input function!

```
Complex *
complex_in(char *str)
{
    double x, y;
    Complex *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2) {
        elog(WARN, "complex_in: error in parsing
        return NULL;
    }
```

```
    result = (Complex *)palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    return (result);
}
```

The output function can simply be:

```
char *
complex_out(Complex *complex)
{
    char *result;
    if (complex == NULL)
        return(NULL);

    result = (char *) palloc(60);
    sprintf(result, “(%g,%g)”, complex->x, complex->y);
    return(result);
}
```

- Try to make the input and output functions inverses of each other. If you do not, you will have severe problems when you need to dump your data into a file and then read it back in (say, into someone else's database on another computer). This is a particularly common problem when floating-point numbers are involved.

To define the `complex` type, you need to create the two user-defined functions `complex_in` and `complex_out` before creating the type:

```
CREATE FUNCTION complex_in(opaque)
RETURNS complex
AS ‘/usr/local/postgres95/tutorial/obj/complex.so’
LANGUAGE ‘c’;

CREATE FUNCTION complex_out(opaque)
RETURNS opaque
AS ‘/usr/local/postgres95/tutorial/obj/complex.so’
LANGUAGE ‘c’;

CREATE TYPE complex (
internallength = 16,
input = complex_in,
output = complex_out
);
```

As discussed earlier, POSTGRES fully supports arrays of base types. Additionally, POSTGRES supports arrays of user-defined types as well. When you define a type, POSTGRES automatically provides support for arrays of that type. For historical reasons, the array type has the same name as the user-defined type with the underscore character `‘_’` prepended to it.

Composite types do not need any function defined on them, since the system already understands what they look like inside.

**A Note About Large Objects**: The types discussed to this point are all "small" objects—that is, they are smaller than 8KB[5] in size. If you require a larger type for something like a document retrieval system or for storing bitmaps, you will need to use the POSTGRES large object interface.

## *2.6 EXTENDING SQL: OPERATORS*

POSTGRES supports left unary, right unary and binary operators. Operators can be overloaded, or reused, with different numbers and types of arguments. If there is an ambiguous situation and the system cannot determine the correct operator to use, it will return an error and you may have to typecast the left and/or right operands to help it understand which operator you meant to use.

The following example shows how to create an operator for adding two complex numbers. First you need to create a function to add the new types. Then, you can create the operator with the function.

```
CREATE FUNCTION complex_add(complex, complex)
RETURNS complex
AS '$PWD/obj/complex.so'
LANGUAGE 'c';

CREATE OPERATOR + (
leftarg = complex,
rightarg = complex,
procedure = complex_add,
commutator = +
);
```

To create unary operators, just omit one of `leftarg` (for left unary) or `rightarg` (for right unary) from the binary operator example. Comparison operators, such as '<', '>' and '=' are defined the same as other binary operators. The type returned by a comparison operator must be boolean (TRUE or FALSE). All of the POSTGRES operators are defined in this way; a function definition and an operator based on the function. For example, the '=' operator is defined for `int4` using a built-in function `int4eq`, for `char16` using `char16eq`, for `float8` using `float8eq`, and so on.

If you give the system enough type information, it can automatically figure out which operators to use.

```
SELECT (a + b) AS c FROM test_complex;
```

| c |
|---|
| (5.2,6.05) |
| (133.42,144.95) |

## *2.7 EXTENDING SQL: AGGREGATES*

Aggregates in POSTGRES are expressed in terms of state transition functions. That is, an aggregate can be defined in terms of state that is modified whenever an instance is

[5] 8 * 1024 = 8192 bytes. In fact, the type must be considerably smaller than 8192 bytes, since the POSTGRES tuple and page overhead must also fit into this 8KB limitation. The actual value that fits depends on the machine architecture.

processed. Some state functions look at a particular value in the instance when computing the new state (sfunc1 in the create aggregate syntax) while others only keep track of their own internal state (sfunc2).

If you define an aggregate that uses only `sfunc1,` you are defining an aggregate that computes a running function of the attribute values from each instance. "Sum" is an example of this kind of aggregate. "Sum" starts at zero and always adds the current instance's value to its running total. The following example uses the `int4pl` that is built into POSTGRES to perform this addition.

```
CREATE AGGREGATE complex_sum (
sfunc1 = complex_add,
basetype = complex,
stype1 = complex,
initcond1 = '(0,0)'
);

SELECT complex_sum(a) FROM test_complex;
```

| complex_sum |
|---|
| (34,53.9) |

If you define only `sfunc2,` you are specifying an aggregate that computes a running function that is independent of the attribute values from each instance. "Count" is the most common example of this kind of aggregate. "Count" starts at zero and adds one to its running total for each instance, ignoring the instance value. Here, you can use the built-in `int4inc` routine to do the work for you. This routine increments (adds one to) its argument.

```
CREATE AGGREGATE my_count (sfunc2 = int4inc, -- add one
basetype = int4, stype2 = int4,
initcond2 = '0');

SELECT my_count(*) as emp_count from EMP;
```

| emp_count |
|---|
| 5 |

"Average" is an example of an aggregate that requires both a function to compute the running sum and a function to compute the running count. When all of the instances have been processed, the final answer for the aggregate is the running sum divided by the running count. You can use the `int4pl` and `int4inc` routines you used previously as well as the POSTGRES integer division routine, `int4div,` to compute the division of the sum by the count.

```
CREATE AGGREGATE my_average (sfunc1 = int4pl, -- sum
basetype = int4,
stype1 = int4,
sfunc2 = int4inc, -- count
stype2 = int4,
finalfunc = int4div, -- division
initcond1 = '0',
initcond2 = '0');

SELECT my_average(salary) as emp_average FROM EMP;
```

| emp_average |
|---|
| 1640 |

## *2.8 INTERFACING EXTENSIONS TO INDICES*

The procedures described to this point enable you to define a new type, new functions, and new operators. However, you have not yet seen how to define a secondary index (such as a B-tree, R-tree or hash access method) over a new type or its operators.

Look back at Figure 2. The right half shows the catalogs that you must modify in order to tell POSTGRES how to use a user-defined type and/or user-defined operators with an index (i.e., `pg_am`, `pg_amop`, `pg_amproc` and `pg_opclass`). Unfortunately, there is no simple command to do this. This section demonstrates how to modify these catalogs through a running example: a new operator class for the B-tree access method that sorts integers in ascending absolute value order.

The `pg_am` class contains one instance for every user- defined access method. Support for the heap access method is built into POSTGRES, but every other access method is described here. The schema is described in Table 4.

| | |
|---|---|
| **amname** | name of the access method |
| **amowner** | object id of the owner's instance in pg_user |
| **amkind** | not used at present, but set to 'o' as a place holder |
| **amstrategies** | number of strategies for this access method (see below) |
| **amsupport** | number of support routines for this access method (see below) |
| **amgettuple , aminsert, ...** | procedure identifiers for interface routines to the access method. For example, `regproc` IDs for opening, closing, and getting instances from the access method appear here. |

*Table 4: pg_am schema*

The object ID of the instance in `pg_am` is used as a foreign key in lots of other classes. You don't need to add a new instance to this class; all you're interested in is the object ID of the access method instance you want to extend:

```
SELECT oid FROM pg_am WHERE amname = 'btree'
```

| oid |
|---|
| 403 |

The `amstrategies` attribute standardizes comparisons across data types. For example, B-trees impose a strict ordering on keys, lesser to greater. Because POSTGRES allows the user to define operators, POSTGRES cannot look at the name of an operator (e.g., `>` or `<`) and tell what kind of comparison it is. In fact, some access methods don't impose any ordering at all. For example, R-trees express a rectangle-containment relationship, whereas a hashed data structure expresses only bitwise similarity based on the value of a hash function. POSTGRES needs some consistent way of taking a qualification in your query, looking at the operator, and then deciding if a usable index exists. This implies that POSTGRES needs to know, for example, that the `<=` and `>` operators partition a B-tree. POSTGRES uses strategies to express these relationships between operators and the way they can be used to scan indices.

Defining a new set of strategies is beyond the scope of this discussion, but we'll explain how B-tree strategies work because you'll need to know that to add a new operator class. In the `pg_am` class, the `amstrategies` attribute is the number of strategies defined for this access method. For B-trees, this number is 5. These strategies correspond to

| | |
|---|---|
| less than | 1 |
| less than or equal | 2 |
| equal | 3 |
| greater than or equal | 4 |
| greater than | 5 |

The idea is that you'll need to add procedures corresponding to the comparisons above to the `pg_amop` relation (see below). The access method code can use these strategy numbers, regardless of data type, to figure out how to partition the B-tree, compute selectivity, and so on. Don't worry about the details of adding procedures yet; just understand that there must be a set of these procedures for `int2, int4, oid,` and every other data type on which a B-tree can operate.

Sometimes, strategies aren't enough information for the system to figure out how to use an index. Some access methods require other support routines in order to work. For example, the B-tree access method must be able to compare two keys and determine whether one is greater than, equal to, or less than the other. Similarly, the R-tree access method must be able to compute intersections, unions, and sizes of rectangles. These operations do not correspond to user qualifications in SQL queries; they are administrative routines used internally by the access methods.

In order to manage diverse support routines consistently across all POSTGRES access methods, `pg_am` includes an attribute called `amsupport.` This attribute records the number of support routines used by an access method. For B-trees, this number is one—the routine to take two keys and return -1, 0, or +1, depending on whether the first key is less than, equal to, or greater than the second. [6]

The `amstrategies` entry in `pg_am` is just the number of strategies defined for the access method in question. The procedures for less than, less equal, and so on don't appear in `pg_am.` Similarly, `amsupport` is just the number of support routines required by the access method. The actual routines are listed elsewhere.

The next class of interest is `pg_opclass.` This class exists only to associate a name with an OID. In `pg_amop,` every B-tree operator class has a set of procedures, one through five, above. Some existing opclasses are `int2_ops, int4_ops,` and `oid_ops.` You need to add an instance with your opclass name (for example, `complex_abs_ops)` to `pg_opclass.` The OID of this instance is a foreign key in other classes.

[6] Strictly speaking, this routine can return a negative number or a non-zero positive number.

```
INSERT INTO pg_opclass (opcname) VALUES ('complex_abs_ops');

SELECT oid, opcname
FROM pg_opclass
WHERE opcname = 'complex_abs_ops';
```

| oid | opcname |
|---|---|
| 17314 | complex_abs_ops |

Note that the OID for your `pg_opclass` instance will be different! You should substitute your value for 17314 wherever it appears in this discussion.

So now you have an access method and an operator class. But you still need a set of operators; the procedure for defining operators was discussed earlier in this manual. For the `complex_abs_ops` operator class on B-trees, the required operators are:

- absolute value
- less-than absolute value
- less-than-or-equal absolute value
- equal absolute value
- greater-than-or-equal absolute value
- greater-than

Suppose the code that implements the functions defined is stored in the file
`/usr/local/mariposa/src/tutorial/complex.c`

Part of the code look like this: (note that you will only show the equality operator for the rest of the examples. The other four operators are very similar. Refer to `complex.c` or `complex.sql` for the details.)

```
#define Mag(c) (c->x*c->x + c->y*c->y)
bool
complex_abs_eq(Complex *a, Complex *b)
{
double amag = Mag(a), bmag = Mag(b);
   return (amag==bmag);
}
```

There are a couple of important things that are happening below.

First, note that operators for less-than, less-than-or-equal, equal, greater-than-or-equal, and greater-than for `int4` are being defined. All of these operators are already defined for `int4` under the names `<`, `<=`, `=`, `>=`, and `>`. The new operators behave differently, of course. In order to guarantee that POSTGRES uses these new operators rather than the old ones, they need to be named differently from the old ones. This is a key point: you can overload operators in POSTGRES, but only if the operator isn't already defined for the argument types. That is, if you have `<` defined for `int4, (int4),` you can't define it again. POSTGRES does not check this when you define your operator, so be careful. To avoid this problem, odd names will be used for the operators. If you get this wrong, the access methods are likely to crash when you try to do scans.

The other important point is that all the operator functions return Boolean values. The access methods rely on this fact. (On the other hand, the support function returns whatever the particular access method expects—in this case, a signed integer.)

The final routine in the file is the "support routine" mentioned when you discussed the `amsupport` attribute of the `pg_am` class. You will use this later on. For now, ignore it.

```
CREATE FUNCTION complex_abs_eq(complex, complex)
RETURNS bool
AS '/usr/local/mariposa/tutorial/obj/complex.so'
LANGUAGE 'c';
```

Now define the operators that use them. As noted, the operator names must be unique among all operators that take two `int4` operands. In order to see if the operator names listed below are taken, you can do a query on `pg_operator:`

```
/*
 * this query uses the regular expression operator (~)
 * to find three-character operator names that end in
 * the character &
 */
SELECT *
FROM pg_operator
WHERE oprname ~ '^..&$'::text;
```

to see if your name is taken for the types you want. The important things here are the procedure (which are the C functions defined above) and the restriction and join selectivity functions. You should use just the ones used below—note that there are different functions for the less-than, equal, and greater-than cases. These must be supplied, or the access method will crash when it tries to use the operator. You should copy the names for `restrict` and `join,` but use the procedure names you defined in the last step.

```
CREATE OPERATOR = (
leftarg = complex,
rightarg = complex,
procedure = complex_abs_eq,
restrict = eqsel,
join = eqjoinsel
)
```

Notice that five operators corresponding to less, less equal, equal, greater, and greater equal are defined.

The final step is to update the `pg_amop` relation. To do this, you need the following attributes:

| | |
|---|---|
| **amopid** | the OID of the pg_am instance for B-tree (== 403, see above) |
| **amopclaid** | the OID of the pg_opclass instance for int4_abs_ops (== whatever you got instead of 17314, see above) |
| **amopopr** | the oids of the operators for the opclass (which we'll get in just a minute) |
| **amopselect, amopnpages** | cost functions |

The cost functions are used by the query optimizer to decide whether or not to use a given index in a scan. Fortunately, these already exist—`btreesel,` which estimates the selectivity of the B-tree, and `btreenpage,` which estimates the number of pages a search will touch in the tree.

You need the OIDS of the operators you just defined. To find them, look up the names of all the operators that take two `int4s,` and pick yours out:

```
SELECT o.oid AS opoid, o.oprname
INTO TABLE complex_ops_tmp
FROM pg_operator o, pg_type t
WHERE o.oprleft = t.oid and o.oprright = t.oid
and t.typname = 'complex';
```

which returns:

| oid | oprname |
|---|---|
| 17321 | < |
| 17322 | <= |
| 17323 | = |
| 17324 | >= |
| 17325 | > |

(Again, some of your OID numbers will almost certainly be different.) In this example, the operators you are interested in are those with OIDS 17321 through 17325. (The values you actually get will probably be different, and you should substitute them for the values below.) Look at the operator names and pick out the ones you just added.

Now you're ready to update `pg_amop` with our new operator class. The operators should be ordered, from less than through greater than, in `pg_amop.` Add the required instances:

```
INSERT INTO pg_amop (amopid, amopclaid, amopopr, amopstrategy,
amopselect, amopnpages)
SELECT am.oid, opcl.oid, c.opoid, 3,
'btreesel'::regproc, 'btreenpage'::regproc
FROM pg_am am, pg_opclass opcl, complex_ops_tmp c
WHERE amname = 'btree' and opcname = 'complex_abs_ops'
and c.oprname = '=';
```

Note the order: "less than" is 1, "less than or equal" is 2, "equal" is 3, "greater than or equal" is 4, and "greater than" is 5.

The last step is registration of the support routine previously described in our discussion of `pg_am.` The OID of this support routine is stored in the `pg_amproc` class, keyed by the access method `oid` and the operator class `oid.` First, you need to register the function in POSTGRES (recall that you put the C code that implements this routine in the bottom of the file where you implemented the operator routines):

```
CREATE FUNCTION int4_abs_cmp(int4, int4)
RETURNS int4
AS '/usr/local/postgres95/tutorial/obj/complex.so'
LANGUAGE 'c';

SELECT oid, proname FROM pg_proc WHERE prname = 'int4_abs_cmp';
```

| oid | proname |
|---|---|
| 17328 | int4_abs_cmp |

(Again, your OID number will probably be different and you should substitute the value you see for the value below.) Recalling that the B-tree instance's OID is 403 and that of `int4_abs_ops` is 17314, you can add the new instance as follows:

```
INSERT INTO pg_amproc (amid, amopclaid, amproc, amprocnum)
VALUES ('403'::oid, -- btree oid
        '17314'::oid, -- pg_opclass tuple
        '17328'::oid, -- new pg_proc oid
        '1'::int2);
```

## 2.9 THE POSTGRES RULE SYSTEM

Production rule systems are conceptually simple, but there are many subtle points involved in actually using them. Consequently, this guide does explain the actual syntax and operation of the POSTGRES rule system here. Instead, you should read [STON90b] to understand some of these points and the theoretical foundations of the POSTGRES rule system before trying to use rules. The discussion in this section is intended to provide an overview of the POSTGRES rule system and point you to helpful references and examples.

The "query rewrite" rule system modifies queries to take rules into consideration, and then passes the modified query to the query optimizer for execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions. The power of this rule system is discussed in [ONG90] as well as [STON90b].

# 3. MARIPOSA

This section introduces the Mariposa architecture in more detail and extends the examples from Section 2.1 to include distribution. We show how to split a Mariposa class into fragments and how to move data manually from one site to another. We show how Mariposa processes queries over remote data using information from the name server.

## *3.1 Mariposa Modules*

Mariposa consists of the following cooperating processes:

1. A single **site manager** daemon, which supervises the backends

2. One or more **backend** database server processes

3. A **client** frontend process.

Users who are familiar with POSTGRES will recognize similarities between the site manager and backend processes and the postmaster and postgres processes on which they were based.

Referring back to Section 1.3 we briefly summarize the example illustrated in Figure 1. The client program issues queries to the Mariposa system. Queries are expressed in a version of Standard Query Language. See Section 2.1 for a description of SQL used with POSTGRES, the single-site database supplied with Mariposa. The query is passed into an available backend by the site manager. Inside the backend, the query is passed through a **parser** and then through an **optimizer**, which creates a query plan. The query plan describes what operations will be performed to process the query, and in what order. The optimizer used in Mariposa is a single-site optimizer. It produces a plan as if all the data resided at a single site. The single-site plan is then passed to the backend's **fragmenter** module. The fragmenter produces a fragmented query plan which reflects the fragmentation of the tables referenced in the query and is "parallelizable" to a greater or lesser degree. The fragmenter is described in Section 3.3. The parser, optimizer and fragmenter use information from a **name server** module, running at the same or a different site.

The fragmenter passes the fragmented plan to the backend's **query broker**. The query broker is explained in detail in Section **Error! Reference source not found.**. A Mariposa user allocates a **budget** to each query. The goal of the query broker is to select sites to process the query within the allotted budget. The query broker decides which Mariposa site will process each node in the query plan by following one of two protocols:

- In the *long protocol*, the query broker contacts the site manager's **bidder** module at each potential processing site. The broker waits for responses from the bidders before selecting the best ones.

- In the *short protocol*, the query broker uses information collected from the name server to select the processing sites, thereby avoiding the cost of contacting many remote sites.

After the query broker has specified the processing sites, the backend's **coordinator** module takes over, notifying the remote sites to begin processing, collecting the results, and returning the answer to the client program.

Each Mariposa server site contains a **bidder** module, which is part of the site manager process. The bidder, explained in Section **Error! Reference source not found.**, responds to requests for bids from the query broker. When a bidder receives a request to bid on part of a query, it may either refuse to bid, or return a bid to the query broker. The bid includes the price to perform the work, and a time bound within which the work must be completed. If a bidder bids, then it must process the query if it is chosen by the query broker to do so.

Winning bids must sooner or later be processed. To execute these queries, the site manager allocates an idle backend to it. The number of backends controls the multiprocessing level at each site, and may be adjusted as conditions warrant. The local backend sends the results of the subquery to the site executing the next part of the query or back to the coordinator process.

Each Mariposa server site also includes a **data broker**. The data broker was not mentioned in the example in Section 1.3. The data broker is called after each query is run, whether the query originated at a remote site or locally. Based on data access patterns and space considerations, it engages in buying and selling fragments with data brokers at other Mariposa sites. See Section 3.5.

The behavior of the bidder and data broker processes are controlled through use of the Tcl scripting language. Using Tcl, it is straightforward to change policy decisions; one simply modifies the rules and scripts by which these modules are implemented.

Note: We have not included an explanation of Tcl. Readers unfamiliar with Tcl can refer to one of the books on the subject (such as *An Introduction to the Tcl and Tk Toolkit* by John Ousterhout, Addison-Wesley 1994) or to the `comp.lang.tcl` USENET newsgroup.

## *3.2 A Distributed Example*

The source for all of the examples in this section can be found in `'src/tutorial/dist.sql'`. This section assumes that you have installed Mariposa on at least two sites. For instructions on setting up Mariposa, refer to the *Installation and Setup Guide*. In this section:

- We assume that you have a site manager process running at the two sites referred to as numbers 1 and 2 in the examples.
- We assume that Site 1 is a name server and has subscribed to the metadata for Site 2 using the SUBSCRIBE METADATA command. See Section 3.4 and the *Installation and Setup Guide.*
- In the example, data is moved from one site to another and then queried immediately, so the update interval for the SUBSCRIBE METADATA

statement should be relatively short, for example 60 seconds. See Section 3.4.1 for an explanation of SUBSCRIBE METADATA.

- Site 1 is the home site, where we are issuing queries. Site 2 is a remote site, in this case, Miami.

### *3.2.1 Creating a Mariposa Class*

In Mariposa, classes are created at a site just as in a single-site database. However, the CREATE TABLE command in Mariposa has been extended. In Mariposa, a user can specify how the table is to be partitioned, should the table ever be split into fragments. Note that when a table is split, it is always split into two fragments. Each of the resulting fragments can be split again, and so on. There are four ways to partition a table in Mariposa, shown in Table 5.

| **Partition Mode** | **Explanation** |
|---|---|
| *Random* | Records are placed in one fragment or the other at random. |
| *Round-Robin* | Half the records are placed in one fragment, half in the other. |
| *Key-Based* | The records are partitioned by value based on one of the attributes of the class and a split value, supplied by the user. |
| *Hash-Based* | The records are partitioned by the comparing the result of a function over one of the attributes to a value. The function, the attribute and the split value are all supplied by the user. |

*Table 5: Mariposa Partitioning Strategies*

The SQL CREATE TABLE statement has been extended in Mariposa to include a PARTITION clause. The syntax for the PARTITION clause for the four partitioning modes is in Table 6.

| **Partition Mode** | **PARTITION Clause Syntax** |
|---|---|
| *Random* | RANDOM |
| *Round-Robin* | RONDROBIN |
| *Key-Based* | PARTITION ON <*attribute*> USING < *function*> |
| *Hash-Based* | PARTITION ON < *function*> ( <*attribute*> ) |

*Table 6: PARTITION Clause Syntax*

The syntax for key-based and hash-based partitioning requires some explanation. The <*attribute*> is one of the columns in the table. <*function*> is a function that takes two arguments and returns -1, 0 or +1 depending on whether the first argument is less than, equal to, or greater than the second. In general, the comparison function is an access method comparison function, as described in Section 2.8. All the tuples in which <*attribute*> is less than or equal to the split value go in the first fragment. All tuples in which <*attribute*> is greater than the split value go in the second fragment.

The widgets in the example in Section 2.1.1 were kept in two warehouses: Miami and New York. We will keep the records for widgets in the location where they are stored, so we will split using key-based partitioning on the LOCATION attribute. We create the WIDGETS class with the following SQL statement:

```
CREATE TABLE WIDGETS (
   PART_NO   int4,
   LOCATION  char16,   -- warehouse: Miami or New York
   ON_HAND   int4,     -- quantity on-hand
   ON_ORDER  int4,     -- quantity on order
   COMMITTED int4      -- quantity sold but not shipped
) PARTITION ON LOCATION USING btchar16cmp;
```

The function `btchar16cmp` takes two arguments of type char16 and returns -1, 0 or +1, as described above. In general, for key- and hash-based partitioning, the b-tree comparison functions work. They are named analogously to `btchar16cmp`: substitute the appropriate type in place of `char16`.

### *3.2.2 Splitting a Class into Fragments*

Now that we have created the WIDGETS class with the proper partitioning mode, the class can be populated just as in Section 2.1.2 with the INSERT statement:

```
INSERT INTO WIDGETS
VALUES (1, 'New York', 500, 1500, 300);
```

Or with the COPY command:

```
COPY WIDGETS FROM 'src/tutorial/widgets.txt';
```

Now we can split the table into two fragments using the SPLIT FRAGMENT statement:

```
SPLIT FRAGMENT WIDGETS
INTO WIDGETS_MI, WIDGETS_NY
AT 'Miami';
```

The SPLIT FRAGMENT statement creates two new relations which together make up the original relation. Using key-based partitioning, all the tuples in which LOCATION <= 'Miami' go in WIDGETS_MI. All the tuples in which LOCATION > 'Miami' go in WIDGETS_NY. This effectively splits the tuples so that the Miami tuples go in WIDGETS_MI and the New York tuples go in WIDGETS_NY. The original relation can be queried like before:

```
SELECT * FROM WIDGETS;
```

| PART_NO | LOCATION | ON_HAND | ON_ORDER | COMMITTED |
|---|---|---|---|---|
| 1 | New York | 500 | 1500 | 300 |
| 2 | New York | 3000 | 0 | 1000 |
| 3 | Miami | 10000 | 5000 | 8000 |
| 4 | Miami | 8500 | 0 | 200 |
| 5 | New York | 2500 | 2000 | 2000 |
| 3 | New York | 1800 | 200 | 750 |
| 2 | Miami | 9300 | 700 | 5000 |
| 4 | New York | 3200 | 0 | 0 |
| 6 | New York | 1800 | 5000 | 1500 |
| 6 | Miami | 11000 | 0 | 3000 |

Each fragment can also be queried individually:

```
SELECT * FROM WIDGETS_MI;
```

| PART_NO | LOCATION | ON_HAND | ON_ORDER | COMMITTED |
|---|---|---|---|---|
| 3 | Miami | 10000 | 5000 | 8000 |
| 4 | Miami | 8500 | 0 | 200 |
| 2 | Miami | 9300 | 700 | 5000 |
| 6 | Miami | 11000 | 0 | 3000 |

```
SELECT * FROM WIDGETS_NY;
```

| PART_NO | LOCATION | ON_HAND | ON_ORDER | COMMITTED |
|---|---|---|---|---|
| 1 | New York | 500 | 1500 | 300 |
| 2 | New York | 3000 | 0 | 1000 |
| 5 | New York | 2500 | 2000 | 2000 |
| 3 | New York | 1800 | 200 | 750 |
| 4 | New York | 3200 | 0 | 0 |
| 6 | New York | 1800 | 5000 | 1500 |

Alternatively, to create the two fragments WIDGETS_MI and WIDGETS_NY and populate them with the appropriate tuples, we could have created the WIDGETS table and split it without populating it first. Then we could have inserted tuples into the fragments, putting the New York tuples in WIDGETS_NY and the Miami tuples in WIDGETS_MI.

### *3.2.3 Moving Fragments*

Now, suppose we want to move the WIDGETS_MI fragment to Miami. Say the Mariposa server running in Miami has a hostid of 2. (For an explanation of hostid's, see the *Installation and Setup Guide*). We use the MOVE FRAGMENT command to move WIDGETS_MI to hostid 2:

```
MOVE FRAGMENT WIDGETS_MI to 2;
```

After the fragment has been moved, we can still query the WIDGETS class, as before:

```
SELECT * FROM WIDGETS;
```

| PART_NO | LOCATION | ON_HAND | ON_ORDER | COMMITTED |
|---|---|---|---|---|
| 1 | New York | 500 | 1500 | 300 |
| 2 | New York | 3000 | 0 | 1000 |
| 3 | Miami | 10000 | 5000 | 8000 |
| 4 | Miami | 8500 | 0 | 200 |
| 5 | New York | 2500 | 2000 | 2000 |
| 3 | New York | 1800 | 200 | 750 |
| 2 | Miami | 9300 | 700 | 5000 |
| 4 | New York | 3200 | 0 | 0 |
| 6 | New York | 1800 | 5000 | 1500 |
| 6 | Miami | 11000 | 0 | 3000 |

If the query SELECT * FROM WIDGETS only retrieves the New York tuples, the name server (in this case, Site 1) hasn't received the metadata from Site 2. If you have set up Site 1 as the name server, as indicated at the beginning of this section, wait until the update interval has passed and issue the query again.

### *3.2.4 Copying a Fragment*

Now, suppose that we want to make a read-only copy of the WIDGETS_MI fragment so that we can query it without the latency of network traffic. We can make a read-only copy by issuing the COPY FRAGMENT command:

```
COPY FRAGMENT READONLY WIDGETS_MI
FROM 2 UPDATE EVERY 3600;
```

This command causes a read-only copy of WIDGETS_MI to be brought from Site 2 to Site 1 (the home site). Furthermore, updates from Site 2 will be brought over and applied every hour (3600 seconds). This means that the copy of WIDGETS_MI at Site 1 is up to an hour out of date. If Site 2 generated writes frequently and users at Site 1 required more accurate information a smaller update interval might have been appropriate.

The next two sections describe the Mariposa replica system and name service in more detail. Section 3.5 describes the Mariposa Data Broker. Section 0 explains distributed query processing in Mariposa in more detail.

## *3.3 THE MARIPOSA REPLICA SYSTEM*

Mariposa permits the replication of data fragments. In the current implementation, a replica is created from one other replica, which we will refer to as its *parent*. Replicas created from a parent are called its *children*. Each replica periodically receives updates from its parent. This allows Mariposa to use one replica in the place of another, improving availability during host crashes and network failures, and improving performance.

There are two types of Mariposa replicas: A *read-only replica* receives all updates from its parent but cannot process updates; A *read-write replica* propagates its updates to its children, as well as receiving updates from its parent, if it has one. We use the term *update* to mean any tuple insertion, deletion, or modification.

Updates are sent from a parent to a child in an *update stream*. Update streams are initiated by the site manager of the parent site at regular intervals. The update interval is specified at the time a copy is created.

There are two update streams for read-write replicas: one from parent to child, and one from child to parent. Read-only replicas only receive update streams from their parents.

### 3.3.1 Creating a Copy

To create a copy of a fragment in the Mariposa system, use the COPY FRAGMENT command at the site requesting a copy (the *child site*). The site that owns the copy will be referred to as the *parent site*. The COPY FRAGMENT command causes a request to be sent to the parent site to send a copy. When the request has been processed, the child site owns a fragment whose contents are the same as the parent as of the time of transfer. A copy contract is set up at each site, which will cause update streams to be sent back and forth.

The name of a copy is generated automatically by Mariposa. The hexadecimal value of the fragment storage id of the copy, which uniquely identifies it, is appended to the first eight characters of the name of the parent fragment. For example, if a copy of the fragment `WIDGET_MI` were made the name of the copy would be something like `WIDGET_M0A3FBC23`.

The syntax for the `COPY FRAGMENT` command is:

```
COPY FRAGMENT [READONLY] fragment_name
FROM hostid
UPDATE EVERY period
```

*fragment_name* is the name of the fragment at the parent site. The parent site's hostid is *`hostid`*. *`period`* is the amount of time, in seconds, that passes between update stream sendoffs. If the `READONLY` option is specified then a read-only copy is made.

### 3.3.2 Dropping a Copy

Dropping a copy in Mariposa removes the contents of the copy from the database and cancels the update contract with the parent site. *Note*: The system will not warn you if you try to drop the last copy of a fragment.

The syntax for the drop copy command is:

```
DROP COPY fragname
```

`where` *`fragname`* is the name of the copy.

### 3.3.3 Moving a Copy

Since copies are fragments with associated update stream contracts, the `MOVE FRAGMENT` command is used to move copies as well as fragments. In the case of copies, the system takes care of renogotiating the update stream contracts between the other copy holders and the new copy holder. See Section 3.2.3 for information on moving fragments.

## 3.4 MARIPOSA NAME SERVICE

The purpose of name service is to supply client sites with the necessary information to run queries on remote data. In order to process a query on remote data, Mariposa needs the queried tables' metadata at various stages:

- During parsing, the syntactic correctness of the query statement has to be verified. This requires information about the queried tables' attributes and their types, about operators used in the query etc.

- The fragmenter needs information about the fragmentation of remote tables.

- The query broker needs to know the location of the remote fragements.

For local tables, this information is stored in the site's local database catalogs. For remote tables, a name server provides the information stored in the remote database catalogs to its clients by replicating the remote catalogs. This replication is achieved by using the copy mechanism described in Section 3.3.

A Mariposa name server is a regular Mariposa site, which keeps read-only copies of a subset of the system catalogs of other sites. These copies are maintained by the update streams sent from the source sites to the name server; as a consequence, information obtained from a name server will always be out-of-date by a certain amount, just like copies of regular data. The set of sites whose system tables are on a name server is not fixed and does not have to include every existing site. The DBA of a site autonomously determines if that site should also provide name service and which remote sites' catalogs it should replicate.

One difference between catalog data and regular user data replication is that the name server site does not keep the catalog tables from each of the remote sites separated in its own local tables. Instead, the data of all the remote sites' tables is merged into a single set of name server catalog tables. In the current version of Mariposa, only the data from the catalog tables pg_class, pg_fragment and pg_attribute are replicated; they are stored in the name server catalog tables pg_nsvcclass, pg_nsvcfrag and pg_nsvcattr.

### *3.4.1 Setting Up Name Service*

Every Mariposa site can be set up to be a Mariposa name server, because every site has the basic infrastructure needed to provide name service: the replication protocol and the name server catalog tables. In order to fill the name server tables, the database administrator has to establish copy contracts with those sites whose system catalogs it wants to replicate. We call these sites the *source sites*. The command

```
SUBSCRIBE METADATA hostID update-interval
```

sets up a read-only copy contract with the source site indicated by *hostID*. The data of the relevant system catalogs is sent from the source to the name server site and is merged into the name server catalog tables. The source site periodically sends an update stream to the name server site, which is also applied to the name server catalog tables.

A variation of the SUBSCRIBE METADATA command is the SUBSCRIBE NAMESERVICE command:

```
SUBSCRIBE NAMESERVICE hostID update-interval
```

The SUBSCRIBE METADATA command allows a name server to acquire the data from the source site's name server catalog tables. This reduces the overhead of running name service, because a name server can reuse the data acquired by other name servers. It would be possible to create a system with one of the name servers having a contract with every existing client site and all the other name servers simply replicating that name server's data.

To stop serving the meta data from a particular site, use the UNSUBSCRIBE METADATA command:

```
UNSUBSCRIBE METADATA hostID
```

This cancels the copy contract with the source site and removes its meta data from the name server's catalog tables. Similarly, to cancel a SUBSCRIBE NAMESERVICE command, use UNSUBSCRIBE NAMSERVICE.

### *3.4.2 Specifying A Primary Name Server*

Every site needs to have access to a name server in order to process queries on remote tables. The primary name server of a site is specified with the command

```
set nameserver hostID
```

This site will cause Mariposa to contact the name server indicated by *hostID* for all name service information. If a site is itself a name server, it could simply supply its own host ID. Note that even if a site is a name server, it may use a remote site for its own name service.

## *3.5 THE MARIPOSA DATA BROKER*

The Mariposa data broker performs data placement, moving and copying fragments, in response to access patterns. The data broker is a Tcl script, which means that it is easily modified by Mariposa users. When the site manager process is started, it looks in the directory $PGDATA/files for a file called databroker.tcl. This file must contain a procedure called DataBroker, written in Tcl, which accepts no arguments and has no return value.

The DataBroker procedure is called each time a query is finished running. The query may have originated locally, or it may be part of a remote query being run on behalf of another site. The site manager defines a few global Tcl variables that the data broker can use to make decisions about moving data around. One variable is called `hostid` and identifies the local hostid. This is necessary for the data broker to examine whether data is stored locally or remotely. The other Tcl variable made available to the data broker is called `fragments`. It is a Tcl list which describes all of the data fragments accessed in the query that just ran[*]. There is one entry in the list for each class accessed in the query. Each entry is of the form:

$\{$`classid logicalid storeid pages tuples` $\{loc_1\ loc_2 ...\ loc_n\}\ \}$

The items are explained in Table 7.

[*] The list will contain information about all fragments of all classes accessed in the query, whether the query originated locally or not. This gives data brokers running at all sites access to information about which fragments are being used most often.

| Item | Description |
|---|---|
| classid | The OID (Object Identifier) of the fragment's class. Shared by all copies of all fragments of a class. |
| logicalid | The OID shared by all copies of the fragment. |
| storeid | The OID of the fragment at the site where it was accessed for the current query. Unique to this copy of the fragment. |
| pages | The size of the fragment, in disk pages. |
| tuples | The size of the fragment, in tuples. |
| $loc_1$ $loc_2$... $loc_n$ | The hostid's of the storage locations of all copies of the fragment. `loc1` is the hostid of the location where the fragment was accessed for the current query. |

*Table 7: Fragment Information for Data Broker*

The data broker may use the information in any way it wants. Here is a small data broker that buys fragments it doesn't already have the second time they're accessed.

```
#########################################################
# databroker.tcl--
##########################################################

set fragmentsAccessed ""

proc DataBroker {} {
    global fragments
    global hostid

global fragmentsAccessed

    # Go through each fragment in the list of fragments
    # accessed.  If a fragment has been accessed twice
    # then buy it.
    #
    # Use the fragment's logicalid to keep track of which
    # fragments have been accessed.  Logicalid's are unique
    # across sites but are the same for copies of the same
    # fragment.
    foreach fragment $fragments {
        set fraStorageSites [lindex $fragment 5]

        # If we don't already have a copy of the fragment at
        # this site
        if {[lsearch $fraStorageSites $hostid] == -1} {
            set fralogicalid [lindex $fragment 1]
            set listPos [lsearch $fragmentsAccessed $fralogicalid]

            # If we've already seen this fragment before, remove it
            # from the list of fragments accessed and bring it to
            # this site using takefragment.  Otherwise, record it.
```

```
                if {$listPos > -1} {
                    set fragmentsAccessed [lreplace $fragmentsAccessed
                                           $listPos $listPos]
                    set fraclassid [lindex $fragment 0]
                    set frastoreid [lindex $fragment 2]
                    set frahostid [lindex $fraStorageSites 0]
                    takefragment $fraclassid $frastoreid $frahostid
                } else {
                    lappend fragmentsAccessed $fralogicalid
            }
        }
    }
```

We have added two Tcl commands for the data broker: `takefragment` and `movefragment`. The syntax for these commands is:

```
takefragment <classid> <storeid> <fromhostid>
movefragment <classid> <storeid> <tohostid>
```

As the names suggest, `takefragment` takes a fragment from the remote host indicated by `<fromhostid>` and installs it locally. `movefragment` moves a fragment to the remote host <tohostid>.

## *3.6 QUERY PROCESSING IN MARIPOSA*

As described in Section 1.3, when a user submits a query to Mariposa, it passes through several modules including a parser, optimizer, fragmenter and so on. The behavior of some of these modules can be affected by the user to a greater or lesser extent either by using special Mariposa commands or, in the case of the bidder, by providing the Tcl script that defines its behavior. This section describes these modules and how a Mariposa user can change their behavior and thus the behavior of the system.

### *3.6.1 The Fragmenter*

After the Mariposa system accepts a query from a client process, parses it and performs optimization, the system hands the query, in the form of a query plan, to the fragmenter. The Mariposa fragmenter module takes a plan that only references whole classes and produces a plan that reflects the underlying fragmentation of the tables.

In this section, we describe how different fragmented plans can be produced from one single-site plan, and how to control the fragmentation process.

#### 3.6.1.1 Fragmented Query Plans

After the fragmenter accepts a query plan from the single-site optimizer, it descends to the leaves of the plan tree, which represent scans over base relations. For example, the query

```
SELECT * from EMP, DEPT where EMP.deptno = DEPT.no;
```

may be converted into the plan tree shown below in Figure 3 by the single-site optimizer before fragmentation. The base relations are unindexed, so a sequential scan (SS) is used. The data from each sequential scan is sorted by department number into a temporary relation, which is scanned before performing the join.

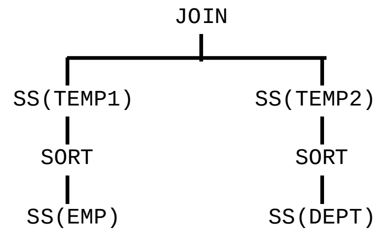


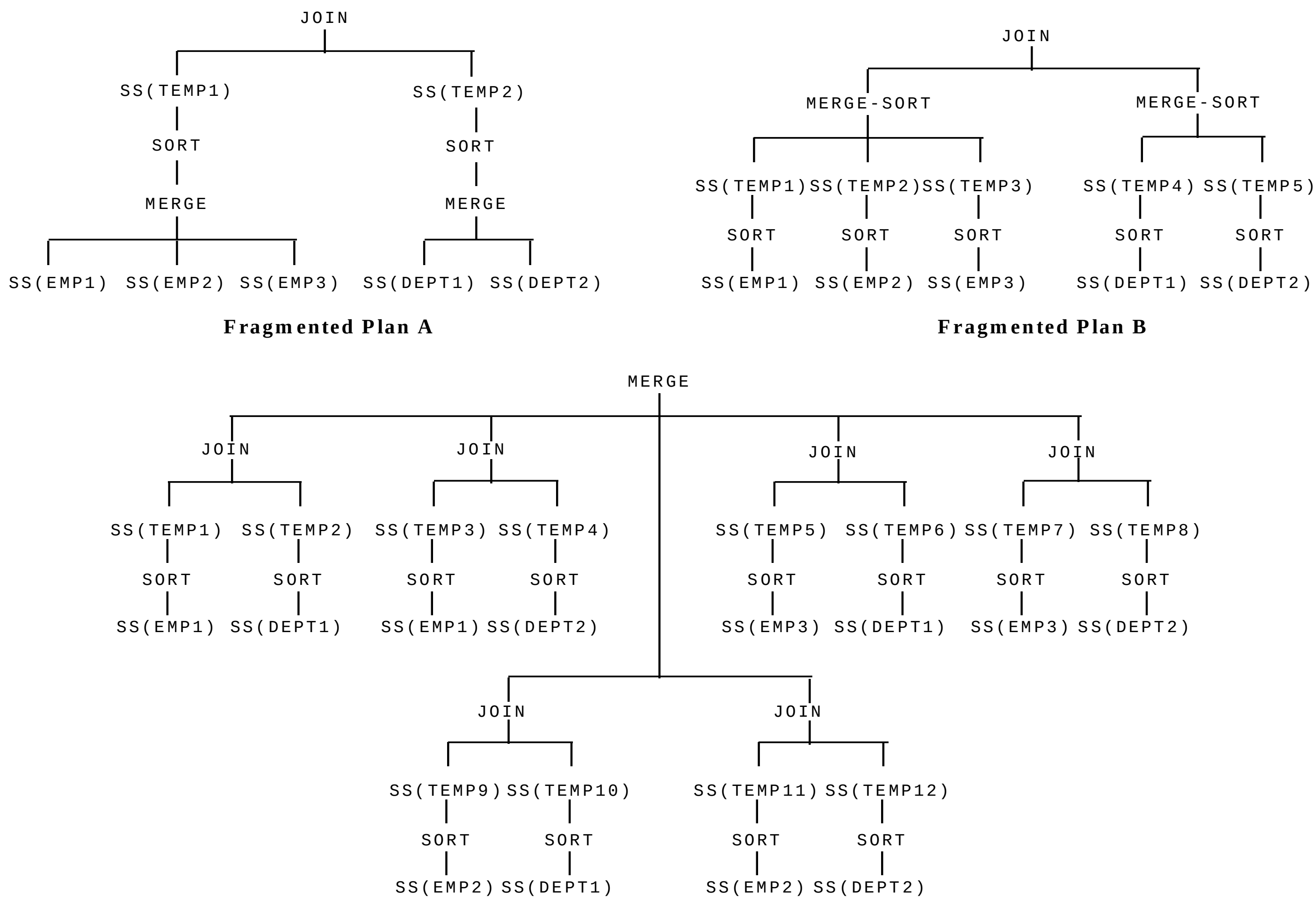

*Figure 3: Unfragmented Query Plan*

As explained in Section 3.2.2, base relations in Mariposa can be partitioned horizontally into data fragments. Each data fragment contains some fraction of the tuples in the base relation. Together, the data fragments represent the entire base relation. Each of the scans in the plan tree is divided into one or more scans, one for each data fragment.

In our example the EMP relation is fragmented into EMP1 EMP2 and EMP3. Similarly, the DEPT relation is fragmented into DEPT1 and DEPT2. The fragmenter starts by dividing the sequential scan of EMP into sequential scans of EMP1, EMP2 and EMP3, and similarly for DEPT. Each of these sequential scans produces a tuple stream. Tuple streams in Mariposa are merged together into a single stream by inserting a *merge node* above them in the plan tree. The merge node takes multiple tuple streams as input and produces one stream as output.

The fragmenter can produce different plans from one unfragmented plan by inserting merge nodes at different places in the plan tree. In our example, any of the fragmented plans shown below in Figure 4 could be produced by the fragmenter. In Fragmented Plan A, the fragmenter has placed merge nodes directly above the fragmented sequential scans. In Fragmented Plan B, the sequential scans are first sorted, then merged together. Since the join in our example requires the input streams to be sorted, the merge nodes must maintain the sorted order, and so are merge-sort nodes in this plan. In Fragmented Plan C, each pair of data fragments is scanned, sorted and joined together, and the resulting streams are merged together.

The placement of merge nodes affects the number of nodes in the fragmented plan and thereby affects the potential for parallel execution of the plan. In Figure 4, Plan A has fewer nodes than Plan B, which has fewer than Plan C. Each of the nodes in Plan A represents more work than each of the nodes in Plan B, with the exception of the sequential scans. Likewise, the nodes in Plan B represent more work, on average, than those in Plan C. If the work is divided between multiple servers, Plan C can achieve a higher degree of parallelism than Plan B, and similarly for Plan B and Plan A.




**Fragmented Plan C**

*Figure 4: The fragmenter can produce different plans from one unfragmented plan by inserting merge nodes at different places in the plan tree.*

The placement of merge nodes in a plan is controlled by the SQL extension:

```
SET FRAGMENTATION <fragmentation factor>;
```

where `<fragmentation factor>` is an integer between 0 and 100 inclusive. As the fragmenter works its way up the plan tree, there are various points at which it may insert a merge node. At each one, the fragmenter generates a random number between 0 and 100. If the number is greater than the fragmentation factor, the fragmenter inserts a merge node. If it is less than the fragmentation factor, it does not. Setting the fragmentation factor to 0 (minimum parallelization) guarantees that the fragmenter will produce a plan like Fragmented Plan A. Setting it to 100 (maximum parallelization) guarantees a plan like Plan C. Setting p to a value between 0 and 100 will result in the fragmenter producing a plan somewhere in between Plan A and Plan C, such as Plan B.

### *3.6.2 The Query Broker*

The Mariposa query broker is responsible for determining the sites at which different pieces of a query will be processed. The query broker attempts to solve a user's query as far under the user's bid curve as possible. First, it divides the query plan up into **plan chunks**. Then it contacts processing sites using either the short or long protocol. If the short protocol is used, the query broker selects the processing site for each plan chunk without contacting the sites first. If the long protocol is used, the query broker solicits bids from several processing sites for each plan chunk, then selects the group of bids that will solve the query and be as far under the bid curve as possible.

In this section, first we explain bid curves. Then, we will discuss plan chunks. Finally, we discuss the short and long protocols.

#### 3.6.2.1 Bid Curves

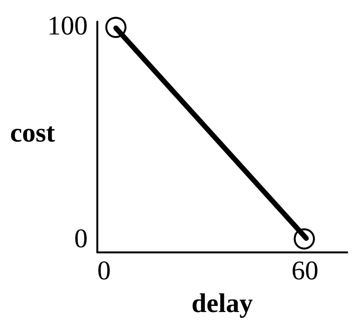
A bid curve is a line in two dimensions: `cost` and `delay`. `Cost` can be in any unit, and `delay` is in seconds. In this discussion, we will use dollars as the cost unit. By defining the bid curve, a Mariposa user specifies how much money he or she will pay to receive an answer within a given amount of time. A user defines a bid curve using the SQL extension `SET BIDCURVE`:

```
SET BIDCURVE cost1, delay1, cost2, delay2
```

`(cost1, delay1)` and `(cost2, delay2)` are two points which define the bid curve. For example, if the user were willing to pay $100 for an answer within five seconds, and nothing for an answer after one minute, he or she would use the command:

```
SET BIDCURVE 100, 5, 0, 60
```

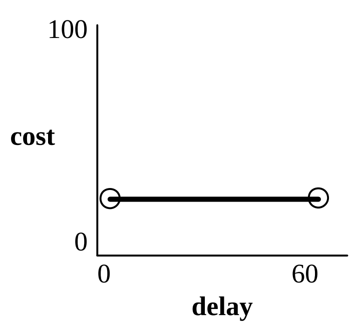
Which would define the curve shown below.




If the user wanted to specify that he or she was willing to pay a maximum of twenty dollars, no matter how long the query took to be processed, the command would be:

```
SET BIDCURVE 20, 0, 20, 60
```

And would result in a bid curve like the one below.



#### 3.6.2.2 Plan Chunks

After the query broker accepts the fragmented plan from the fragmenter, the first thing it does is to divide the fragmented plan into a set of non-overlapping subplans, called *plan chunks*. Each plan chunk is sent out whole to potential processing sites. Dividing a plan into many small chunks increases the potential for parallel and pipelined execution of the plan, while dividing it into a few large chunks decreases potential parallelism and pipelining.

Continuing with the example from Section 3.6.1, if the fragmenter produced Fragmented Plan B from Figure 4, Figure 5 shows three of the possible groups of plan chunks the query broker could produce. In Plan B-1, each plan chunk consists of exactly one node in the plan tree. Therefore, each node in Plan B-1 could be processed by a different server. In Plan B-2, the plan chunks are larger; the sequential scan nodes are grouped with the sort nodes and the sequential scans on temporary relations. The join node is grouped with the merge-sort nodes. In Plan B-3, the entire query is grouped into one large plan chunk. Therefore, the entire query will be sent to potential processing sites.

The size of plan chunks produced by the query broker can be set by the command

```
set chunksize <chunk-factor>;
```

where `<chunk-factor>` is an integer between 0 and 100 inclusive. As the query broker moves from the leaves of the fragmented plan tree towards the root, it generates a random number between 0 and 100 at each step. If the number generated is greater than `<chunk-factor>`, the plan is cut off at that point and a plan chunk is created. If it is less than or equal, the process continues. If `chunksize` is set to 100 (coarsest) the user is guaranteed to get a plan like B-3 in Figure 5. If granularity is set to 0 (finest) the user will get a plan like B-1. If it is set in between, the user will get a plan similar to B-2.

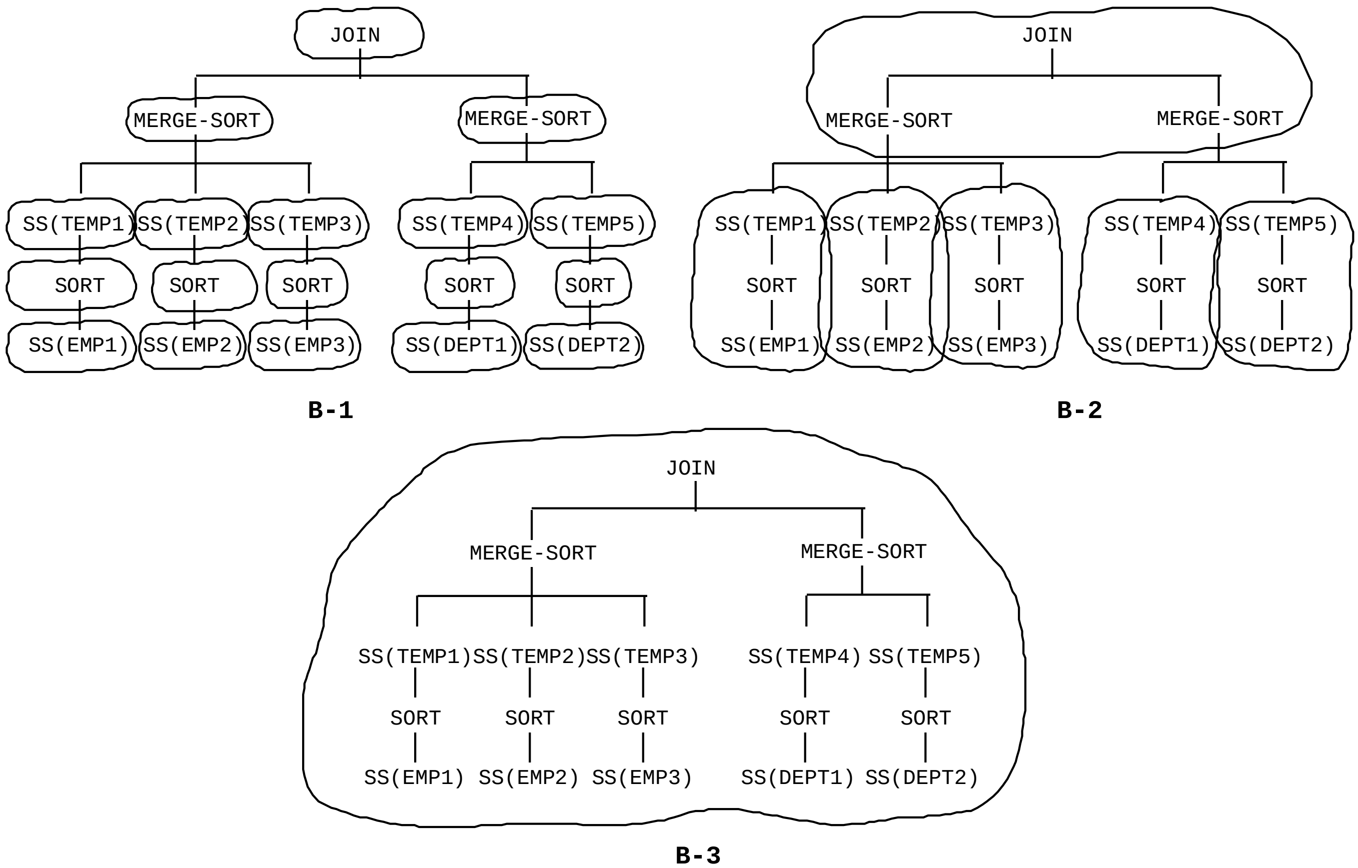



*Figure 5: Plan Chunks*

### 3.6.2.3 Bid Protocols

The query broker may follow either the short or long bidding protocol to determine the site(s) at which a query will be executed. The bid protocol used by the query broker is set with the `SET bidproto` command:

```
SET bidproto 'short'|'long';
```

#### 3.6.2.3.1 The Short Protocol

In the short protocol, the query broker sends each plan chunk to a *single* potential processing site. In this case, the query broker attempts to select the site most likely to have won the bidding process had the long protocol been used. The query broker determines this based on advertising information and statistics it maintains about previous queries.

Once the query broker has determined which sites it will contact, it returns the query plan back to the coordinator, indicating which site is to be contacted to process each plan chunk. The processing site may respond in one of two ways: either by processing the subquery represented by the plan chunk, or by refusing to do so. In the current Mariposa implementation, processing sites always agree to perform work requested by the query broker.

#### 3.6.2.3.2 The Long Protocol

In the long protocol, the query broker sends each plan chunk to a set of bidder sites, which are potential processing sites. Each bidder site responds with a bid, which specifies the cost and delay required to process the subquery. The query broker selects the best bid for each plan chunk and notifies the losing sites. It then sends the query plan back to the coordinator as in the short protocol, indicating the processing site for each plan chunk.

### *3.6.3 The Bidder*

The Mariposa bidder module accepts requests from query brokers to bid on work. Its job is to determine the amount that the site will charge to process the given query plan chunk and the expected processing time (delay).

#### 3.6.3.1 Bidding

The bidder's behavior is controlled entirely by a Tcl script, much like the data broker. When the site manager process starts up, it looks for a file called `bidder.tcl` in the directory $PGDATA/files. `bidder.tcl` contains the procedure `GetQueryBid`, which should take no arguments and return a Tcl list of five elements: `response, price delay staleness and accuracy`, as explained in Table 8.

| Element | Description |
|---|---|
| response | 0 or 1. 1 = Bid. 0 = Refuse to Bid |
| price | The price, in dollars, that this site will charge to process the query |
| delay | The time, in seconds, for the site to process the query from the time it starts processing. Delay does not include network time. |
| staleness | Reserved for future use. For now, it is sufficient to return the value 0.0 |
| accuracy | Reserved for future use. For now, it is sufficient to return the value 0.0 |

*Table 8: The bidder's response*

To change the bidding policy, simply redefine the procedure GetQueryBid. To force the bidder to reload `bidder.tcl`, issue the Tcl command `ReInitBidder` from the site manager's `TclDMT>` prompt.

We have provided some global variables, available to `GetQueryBid`, which may be useful in formulating the bid. These are described in Table 9.

| Variable Name | Value |
|---|---|
| hostid | An integer identifying the ID of the machine the bidder is running on. |
| contract | The unique ID assigned to this contract by the Site Manager. |
| plan | A string representing the plan tree. |
| rtable | A Tcl list which contains information about the relations and fragments accessed in the plan. |

*Table 9. These global variables are made available to `GetQueryBid`.*

`GetQueryBid` may define other global variables and store data in them. These globals will hold their values across future calls to `GetQueryBid` and other procedures in this TclDMT interpreter.

#### 3.6.3.2 The `plan` and `rtable` global variables

The `plan` and `rtable` variables require some explanation. The variable `plan` is a string that represents the plan tree for which the bidder is being asked to formulate a bid. It is a recursive list of the form:

```
{ NODETYPE NODENUM {LEFTTREE} {RIGHTTREE} }
```

where NODETYPE is a string representing the operation, NODENUM is a unique identifier for the node, and LEFTTREE and RIGHTTREE are the left and right subplans, respectively. LEFTTREE and RIGHTTREE may be empty. If RIGHTTREE is empty, LEFTTREE is also empty and the node is a leaf node. If there is a single subplan, it will be in LEFTTREE.

There is one exception to the above format. Merge nodes, which were first discussed in Section , may have more than two children. The format for merge nodes is:

```
{ NODETYPE NODENUM { {CHILD1} {CHILD2}... {CHILDn} } }
```

where NODETYPE is "MERGE". The different node types and their explanations are listed in Table 10.

| **Node Name** | **Explanation** | **Node Format** |
|---|---|---|
| MERGEJOIN | Merge-Join | { MERGEJOIN NODENUM {LEFT-TREE} {RIGHT TREE} } |
| NESTEDLOOP | Nested-Loop Join | { NESTEDLOOP NODENUM {LEFT-TREE} {RIGHT TREE} } |
| SEQSCAN | Sequential Scan | If over a base relation:<br>{SEQSCAN NODENUM RTABLE-INDEX FRAG-INDEX {LEFT-TREE} }<br>If over a temporary relation:<br>{ SEQSCAN NODENUM -1 {LEFT-TREE} } |
| SORT | Sort | { SORT NODENUM {LEFT-TREE} } |
| MERGE | Merge | { MERGE NODENUM { $CHILD_1$} { $CHILD_2$} ... { $CHILD_n$ }} |
| XIN | Exchange-In* | { XIN NODENUM {LEFT-TREE} } |
| AGG | Aggregate, such as count() | { AGG NODENUM {LEFT-TREE} } |
| GROUPBY | Group-By node | { GROUPBY NODENUM {LEFT-TREE} } |
| UNKNOWN | Mariposa will fill in UNKNOWN if the node type isn't one it recognizes. | { UNKNOWN NODENUM {LEFT-TREE} {RIGHT-TREE} } |

*Table 10: Plan Nodes in* `plan` *variable made available to Bidder*

The `rtable` variable is a list of information about the tables referred to in the query plan. `rtable` is short for range table, which comes from the SQL syntax `“range of E is EMP”.` The range table provides information about the size, location and fragmentation information of the tables referred to in the query plan. It is in the form:

```
{ rangeTblEntry rangeTblEntry... }
```

where each `rangeTblEntry` is a list:

```
{ relname refname relid fragInfo }
```

and `fragInfo` is a list in which each entry is in the form:

```
{ fralogicalid frastoreid frapages fratuples storagesites }
```

and (finally) `storagesites` is a list in which each entry is of the form:

```
{ port address hostid }
```

These entries are described in Table 11.

* These are added to plan trees in between nodes processed at different sites. An Exchange-In node accepts a tuple stream from a remote site and feeds it into the next node.

| Entry Name | Description |
|---|---|
| relname | The name of the class. |
| refname | The name used to refer to the class in the query. |
| relid | The OID of the class |
| fralogicalid | The OID shared by all copies of this fragment. |
| frastoreid | The OID for this copy of this fragment |
| frapages | Size of fragment in pages. |
| fratuples | Size of fragment in tuples. |
| port | TCP port (not used) |
| address | Network address of storage site |
| hostid | Hostid of storage site |

*Table 11:Entries in `rtable` variable made available to Bidder*

### 3.6.3.3 The `subcontract` Command

In some cases, a bidder will be asked to bid on some operation that it cannot perform. It can refuse to bid, as mentioned earlier, or it can subcontract out some or all of the work to another processing site. For example, if a bidder is asked to perform a join between two classes, A and B, and it has A but not B, it may choose to subcontract out the sequential scan of B to another site. We have added the command `subcontract` to the Tcl provided with Mariposa. The format of the `subcontract` command is:

```
subcontract <plan> <contract>
```

The subcontract returns a list of five elements: `response`, `price`, `delay`, `staleness` and `accuracy`, as described in Table 8. The `<plan>` passed to `subcontract` is any plan variable. It can be the entire string that was passed into the bidder, or a part of the string representing a subplan. The `<contract>` passed to `subcontract` is the global variable made available to the bidder and described in Table 9. The Tcl bidder script needs to pass it back to identify the larger plan to which the subplan belongs.

When a bidder uses the `subcontract` command, the subplan is passed to the query broker at the bidder's site, which contacts potential processing sites, gets their bids, and returns the best one to the bidder. The bidder can then add the subcontracted price and delay into its own bid and return a completed bid to the query broker that contacted it originally. If the bidder site is awarded the bid, the site manager automatically sends out the subcontracted part of the plan to the appropriate site.

### 3.6.3.4 Sample Bidder Script

A sample bidder script is included in Appendix A. The first procedure, `GetQueryBid`, must be included in all bidder scripts, as mentioned above. This bidder script includes one procedure for each node type. `GetQueryBid` calls `CostBasedBid` to calculate the cost and delay (this bidder script ignores staleness and accuracy). CostBasedBid takes the first element in the plan string passed in, which is the node type, and calls the procedure of the same name. This bidder formulates a bid by recursively visiting the nodes of the plan tree and assigning a cost and delay to each node. The final bid returned by `CostBasedBid` is the sum of the bids for all the nodes in the tree. `GetQueryBid` multiplies the cost element in the bid returned by `CostBasedBid` by the load average. In addition to the procedures corresponding to the node types, there are two utility procedures: `CombineBids` "adds up" two bids and `LoadAverage` returns the 5-, 30- and 60-second load averages.

The procedures which calculate a bid for each node type are similar in structure and function. Each one first passes its children nodes to `GetQueryBid` and gets a bid back. Then the delay and cost are calculated on a per-tuple and per-page basis. In addition to calculating the cost and delay, each procedure also updates the values of the global variables `nTuples`, and `nPages`. `nTuples` is an estimate of the number of tuples processed by the query node for which the procedure was called. `nPages` is the same thing for the number of pages processed. These two variables are internal to the example bidder - they are not part of the required bidder interface, like `delay` and `cost`.

The `SEQSCAN` procedure is more complicated than the others and we discuss it in more detail. Like the other procedures, it calculates cost and delay on a per-tuple and per-page basis. However, in addition to the arguments `nodeNum` and `leftTree`, `SEQSCAN` takes two additional arguments: `scanIndex` and `fragIndex`. `scanIndex` indicates the element in the range table corresponding to the relation being scanned. `fragIndex` indicates the element in the `fragInfo` list of the range table entry for the fragment being scanned. They are used by `SEQSCAN` to access the correct entries in the global variable rtable. `SEQSCAN` uses this information to tell whether the scan is over a base relation or a temporary relation. If the scan is over a base relation, the information contains fragment storage locations, and number of tuples and number of pages in the fragment.

If `SEQSCAN` is called for a temporary relation, it behaves like the other procedures: it calculates the cost and delay, then passes its child node to `GetQueryBid`.

If `SEQSCAN` is called for a base relation, it first checks to see if there is a copy of the fragment stored locally. If so, it gets the number of tuples and number of pages in the fragment from the range table. If there is no local copy, SEQSCAN calls `subcontract`. `Subcontract` is a Tcl extension added for Mariposa and is discussed in the next section.

# 4. ADMINISTERING POSTGRES AND MARIPOSA

This section explains how to run the Mariposa processes, create a Mariposa database, add and delete users, and perform other administrative functions. This section assumes that you have already installed Mariposa on each machine on which it will run. If you have not installed Mariposa, refer to the Installation and Setup Guide.

Even if you are not the administrator of your database, you will find it useful to be familiar with many of these tasks.

## *4.1 Frequent Tasks*

This section discusses frequently-performed administrative tasks..

### *4.1.1 Starting the Site Manager*

If you did not install POSTGRES exactly as described in the installation instructions, you may have to perform some additional steps before starting the `postmaster` process.

- Even if you were not the person who installed POSTGRES, you should understand the installation instructions. The installation instructions explain some important issues with respect to where POSTGRES places some important files, proper settings for environment variables, etc. that may vary from one version of POSTGRES to another.

- You must start the `postmaster` process with the user-id that owns the installed database files. In most cases, if you have followed the installation instructions, this will be the user "postgres". If you do not start the `postmaster` with the right user-id, the backend servers that are started by the `postmaster` will not be able to read the data.

- Make sure that `/usr/local/postgres95/bin` is in your shell command path, because the `postmaster` will use your `PATH` to locate POSTGRES commands.

- Remember to set the environment variable `PGDATA` to the directory where the POSTGRES databases are installed. (This variable is more fully explained in the POSTGRES installation instructions.)

- If you do start the `postmaster` using non-standard options, such as a different TCP port number, remember to tell all users so that they can set their `PGPORT` environment variable correctly.

### *4.1.2 Shutting Down the Postmaster*

If you need to halt the `postmaster` process, you can use the UNIX `kill(1)` command. Some people habitually use the `-9` or `-KILL` option; this should never be necessary and the POSTGRES group does not recommend that you do this, because the `postmaster` will be unable to free its various shared resources, its child processes will be unable to exit gracefully, etc.

### *4.1.3 Adding and Removing Users*

The `createuser` and `destroyuser` commands enable and disable access to POSTGRES by specific users on the host system.

### *4.1.4 Periodic Upkeep*

The `vacuum` command should be run on each database periodically. This command processes deleted instances[7] and, more importantly, updates the system statistics concerning the size of each class. If these statistics are permitted to become out-of-date and inaccurate, the POSTGRES query optimizer may make extremely poor decisions with respect to query evaluation strategies. Therefore, you should run `vacuum` every night or so (perhaps in a script that is executed by the UNIX `cron(1)` or `at(1)` commands).

Perform frequent backups. That is, you should either back up your database directories using the POSTGRES copy command and/or the UNIX `dump(1)` or `tar(1)` commands. You may think, "Why am I backing up my database? What about crash recovery?" One side effect of the POSTGRES "no overwrite" storage manager is that it is also a "no log" storage manager. That is, the database log stores only abort/commit data, and this is not enough information to recover the database if the storage medium (disk) or the database files are corrupted! In other words, if a disk block goes bad or POSTGRES happens to corrupt a database file, you cannot recover that file. This can be disastrous if the file is one of the shared catalogs, such as `pg_database.`

### *4.1.5 Tuning*

Once your users start to load a significant amount of data, you will typically run into performance problems. POSTGRES is not the fastest DBMS in the world, but many of the worst problems encountered by users are due to their lack of experience with any DBMS. Some general tips include:

- Define indices over attributes that are commonly used for qualifications. For example, if you often execute queries of the form

```
SELECT * from EMP where salary < 5000
```

  then a B-tree index on the `salary` attribute will probably be useful. If scans involving equality are more common, as in

```
SELECT * from EMP where salary = 5000
```

[7] This may mean different things depending on the archive mode with which each class has been created. However, the current implementation of the `vacuum` command does not perform any compaction orclustering of data. Therefore, the UNIX files that store each POSTGRES class never shrink and the space reclaimed by `vacuum` is never actually reused.

then you should consider defining a hash index on `salary.` You can define both, though it will use more disk space and may slow down updates a bit. Scans using indices are much faster than sequential scans of the entire class.

- Run the `vacuum` command frequently. This command updates the statistics that the query optimizer uses to make intelligent decisions; if the statistics are inaccurate, the system will make inordinately stupid decisions with respect to the way it joins and scans classes.

- When specifying query qualfications (i.e., the `where` part of the query), try to ensure that a clause involving a constant can be turned into one of the form `range_variable operator constant`, e.g.,

  ```
  EMP.salary = 5000
  ```

  The POSTGRES query optimizer will only use an index with a constant qualification of this form. It doesn't hurt to write the clause as

  ```
  5000 = EMP.salary
  ```

  if the operator (in this case, `=)` has a commutator operator defined so that POSTGRES can rewrite the query into the desired form. However, if such an operator does not exist, POSTGRES will never consider the use of an index.

- When joining several classes together in one query, try to write the join clauses in a "chained" form, e.g.,

  ```
  where A.a = B.b and B.b = C.c and ...
  ```

  Notice that relatively few clauses refer to a given class and attribute; the clauses form a linear sequence connecting the attributes, like links in a chain. This is preferable to a query written in a "star" form, such as

  ```
  where A.a = B.b and A.a = C.c and ...
  ```

  Here, many clauses refer to the same class and attribute (in this case, `A.a).` When presented with a query of this form, the POSTGRES query optimizer will tend to consider far more choices than it should and may run out of memory.

- If you are really desperate to see what query plans look like, you can run the `postmaster` with the `- d` option and then run `monitor` with the `-t` option. The format in which query plans will be printed is hard to read but you should be able to tell whether any index scans are being performed.

## *4.2 Infrequent Tasks*

At some time or another, every POSTGRES site administrator has to perform all of the following actions.

### *4.2.1 Cleaning Up After Crashes*

The `postgres` server and the `postmaster` run as two different processes. They may crash separately or together. The housekeeping procedures required to fix one kind of crash are different from those required to fix the other.

The message you will usually see when the backend server crashes is:

```
FATAL: no response from backend: detected in ...
```

This generally means one of two things: there is a bug in the POSTGRES server, or there is a bug in some user code that has been dynamically loaded into POSTGRES. You should be able to restart your application and resume processing, but there are some considerations:

- POSTGRES usually dumps a core file (a snapshot of process memory used for debugging) in the database directory

  ```
  /usr/local/postgres95/data/base/<database>/core
  ```

  on the server machine. If you don't want to try to debug the problem or produce a stack trace to report the bug to someone else, you can delete this file (which is probably around 10MB). (2) When one backend crashes in an uncontrolled way (i.e., without calling its built-in cleanup routines), the `postmaster` will detect this situation, kill all running servers and reinitialize the state shared among all backends (e.g., the shared buffer pool and locks). If your server crashed, you will get the "no response" message shown above. If your server was killed because someone else's server crashed, you will see the following message:

  ```
  I have been signalled by the postmaster.
  Some backend process has died unexpectedly and possibly
  corrupted shared memory. The current transaction was
  aborted, and I am going to exit. Please resend the
  last query.—The postgres backend
  ```

- Sometimes shared state is not completely cleaned up. Frontend applications may see errors of the form:

  ```
  WARN: cannot write block 34 of myclass [mydb] blind
  ```

  In this case, you should kill the postmaster and restart it.

- When the system crashes while updating the system catalogs (e.g., when you are creating a class, defining an index, retrieving into a class, etc.) the B-tree indices defined on the catalogs are sometimes corrupted. The general (and non-unique) symptom is that all queries stop working. If you have tried all of the above steps and nothing else seems to work, try using the `reindexdb` command. If `reindexdb` succeeds but things still don't work, you have another problem; if it fails, the system catalogs themselves were almost certainly corrupted and you will have to go back to your backups.

  The `postmaster` does not usually crash (it doesn't do very much except start servers) but it does happen on occasion. In addition, there are a few cases where it encounters problems during the reinitialization of shared resources. Specifically, there are race conditions where the operating system lets the `postmaster` free shared resources but then will not permit it to reallocate the same amount of shared resources (even when there is no contention).

  You will typically have to run the `ipcclean` command if system errors cause the `postmaster` to crash. If this happens, you may find (using the UNIX `ipcs(1)` command) that the "postgres" user has shared memory and/or semaphores allocated even though no `postmaster` process is running. In this case, you should run `ipcclean` as the "postgres" user in order to deallocate these resources. Be warned that all such resources owned by the "postgres" user will be deallocated. If you have multiple `postmaster` processes running on the same machine, you should kill all of them before running `ipcclean` (otherwise, they will crash on their own when their shared resources are suddenly deallocated).

### *4.2.2 Moving Database Directories*

By default, all POSTGRES databases are stored in separate subdirectories under `/usr/local/postgres95/data/base.`[8] At some point, you may find that you wish to move one or more databases to another location (e.g., to a filesystem with more free space).

If you wish to move all of your databases to the new location, you can simply:

1. Kill the postmaster.

2. Copy the entire `data` directory to the new location. (Making sure that the new files are owned by user "postgres").

```
% cp -rp /usr/local/postgres95/data /new/place/data
```

1. Reset your `PGDATA` environment variable (as described earlier in this manual and in the installation instructions).

```
# using csh or tcsh...
% setenv PGDATA /new/place/data

# using sh, ksh or bash...
% PGDATA=/new/place/data; export PGDATA
```

1. Restart the `postmaster.`

```
% postmaster &
```

1. After you run some queries and are sure that the newly-moved database works, you can remove the old `data` directory.

```
% rm -rf /usr/local/postgres95/data
```

To install a single database in an alternate directory while leaving all other databases in place, do the following:

1. Create the database (if it doesn't already exist) using the createdb command. In the following steps assume the database is named `foo.`

2. Kill the `postmaster`.

3. Copy the directory `/usr/local/postgres95/data/base/foo` and its contents to its ultimate destination. It should still be owned by the "postgres" user.

```
% cp -rp /usr/local/postgres95/data/base/foo /new/place/foo
```

1. Remove the directory /usr/local/postgres95/data/base/foo:

```
% rm -rf /usr/local/postgres95/data/base/foo
```

1. Make a symbolic link from `/usr/local/postgres95/data/base` to the new directory:

[8] Data for certain classes may be stored elsewhere if a nonstandard storage manager was specified when the classes were created. Use of nonstandard storage managers is an experimental feature that is not supported outside of Berkeley.

```
% ln -s /new/place/foo /usr/local/postgres95/data/base/foo
```

1. Restart the `postmaster.`

### *4.2.3 Updating Databases*

POSTGRES is a research system. In general, POSTGRES may not retain the same binary format for the storage of databases from release to release. Therefore, when you update your POSTGRES software, you will probably also have to modify your databases. This is a common occurrence with commercial database systems as well. Unfortunately, unlike commercial systems, POSTGRES does not come with user-friendly utilities to make your life easier when these updates occur.

In general, you must do the following to update your databases to a new software release:

- Extensions (such as user-defined types, functions, aggregates, etc.) must be reloaded by re-executing the SQL CREATE commands. See Appendix A for more details.

- Data must be dumped from the old classes into ASCII files (using the COPY command), the new classes created in the new database (using the CREATETABLE command), and the data reloaded from the ASCII files.

- Rules and views must also be reloaded by re-executing the various CREATE commands.

You should give any new release a trial period; in particular, do not delete the old database until you are satisfied that there are no compatibility problems with the new software. For example, you do not want to discover that a bug in a type's “input” (conversion from ASCII) and “output” (conversion to ASCII) routines prevents you from reloading your data after you have destroyed your old databases. (This should be standard procedure when updating any software package, but some people try to economize on disk space without applying enough foresight.)

## ***4.3 Database Security***

Most sites that use POSTGRES are educational or research institutions and are generally not greatly concerned about security in their POSTGRES installations. If desired, you can install POSTGRES with additional security features, such as the MIT Kerberos network authentication system. Naturally, such features come with additional administrative overhead that must be dealt with.

### *4.3.1 Kerberos*

POSTGRES can be configured to use the MIT Kerberos network authentication system. This prevents outside users from connecting to your databases over the network without the correct authentication information.

## ***4.4 Querying the System Catalogs***

From time to time, you may want to find out what extensions have been added to a given database. The queries listed below are “canned” queries that you can run on any database to get simple answers. Before executing any of the queries below, be sure to execute the POSTGRES `vacuum` command. (The queries will run much more quickly that way.) Also, note that these queries are also listed in

```
/usr/local/postgres95/tutorial/syscat.sql
```

(You can use cut-and-paste (or the `\i` command) instead of doing a lot of typing.)

This query prints the names of all database administrators and the name of their database(s).

```
SELECT usename, datname
FROM pg_user, pg_database
WHERE usesysid = int2in(int4out(datdba))
ORDER BY usename, datname;
```

This query lists all user-defined classes in the database.

```
SELECT relname
FROM pg_class
WHERE relkind = ‘r’—not indices
and relname !~ ‘^pg_’—not catalogs
and relname !~ ‘^Inv’—not large objects
ORDER BY relname;
```

This query lists all simple indices (i.e., those that are not defined over a function of several attributes).

```
SELECT bc.relname AS class_name,
ic.relname AS index_name,
a.attname
FROM pg_class bc, -- base class
pg_class ic, -- index class
pg_index i,
pg_attribute a—att in base
WHERE i.indrelid = bc.oid
and i.indexrelid = ic.oid
and i.indkey[0] = a.attnum
and a.attrelid = bc.oid
and i.indproc = ‘0’::oid—no functional indices
ORDER BY class_name, index_name, attname;
```

This query prints a report of the user-defined attributes and their types for all user-defined classes in the database.

```
SELECT c.relname, a.attname, t.typname
FROM pg_class c, pg_attribute a, pg_type t
WHERE c.relkind = ‘r’—no indices
and c.relname !~ ‘^pg_’—no catalogs
and c.relname !~ ‘^Inv’—no large objects
and a.attnum > 0 -- no system att’s
and a.attrelid = c.oid
and a.atttypid = t.oid
ORDER BY relname, attname;
```

This query lists all user-defined base types (not including array types).

```
SELECT u.usename, t.typname
FROM pg_type t, pg_user u
WHERE u.usesysid = int2in(int4out(t.typowner))
and t.typrelid = ‘0’::oid—no complex types
and t.typelem = ‘0’::oid—no arrays
and u.usename <> ‘postgres’
ORDER BY usename, typname;
```

This query lists all left-unary (post-fix) operators.

```
SELECT o.oprname AS left_unary,
right.typname AS operand,
result.typname AS return_type
FROM pg_operator o, pg_type right, pg_type result
WHERE o.oprkind = ‘l’—left unary
and o.oprright = right.oid
and o.oprresult = result.oid
ORDER BY operand;
```

This query lists all right-unary (pre-fix) operators.

```
SELECT o.oprname AS right_unary,
left.typname AS operand,
result.typname AS return_type
FROM pg_operator o, pg_type left, pg_type result
WHERE o.oprkind = ‘r’—right unary
and o.oprleft = left.oid
and o.oprresult = result.oid
ORDER BY operand;
```

This query lists all binary operators.

```
SELECT o.oprname AS binary_op,
left.typname AS left_opr,
right.typname AS right_opr,
result.typname AS return_type
FROM pg_operator o, pg_type left, pg_type right, pg_type
result
WHERE o.oprkind = ‘b’—binary
and o.oprleft = left.oid
and o.oprright = right.oid
and o.oprresult = result.oid
ORDER BY left_opr, right_opr;
```

This query returns the name, number of arguments (parameters) and return type of all user-defined C functions. The same query can be used to find all built-in C functions if you change the “C” to “internal”, or all SQL functions if you change the “C” to “postquel”.

```
SELECT p.proname, p.pronargs, t.typname
FROM pg_proc p, pg_language l, pg_type t
WHERE p.prolang = l.oid
and p.prorettype = t.oid
and l.lanname = ‘c’
ORDER BY proname;
```

This query lists all of the aggregate functions that have been installed and the types to which they can be applied. `count` is not included because it can take any type as its argument.

```
SELECT a.aggname, t.typname
FROM pg_aggregate a, pg_type t
WHERE a.aggbasetype = t.oid
ORDER BY aggname, typname;
```

This query lists all of the operator classes that can be used with each access method as well as the operators that can be used with the respective operator classes.

```
SELECT am.amname, opc.opcname, opr.oprname
FROM pg_am am, pg_amop amop, pg_opclass opc, pg_operator opr
WHERE amop.amopid = am.oid
```

```
and amop.amopclaid = opc.oid
and amop.amopopr = opr.oid
ORDER BY amname, opcname, oprname;
```

# Appendices

## *A. Sample Bidder Script*

This bidder script is in '$PGDATA/base/files/bidder.tcl'. It gives an idea of how a cost-based bidder might be constructed. It looks at each node in the query plan passed in and charges a fixed amount at each node per page and/or per tuple. It multiplies the cost element of the bid by the current load average.

```
###########################################################################
# bidder.tcl
#
#   Input:  plan tree, represented as a string
#   Output: list containing {response cost delay staleness accuracy}
#
#              response: BID if all data fragments references in the
#                        query are local.  REFUSETOBID otherwise.
#
#              cost:    Based on the per-tuple and per-page charge for
#                        each node in the query plan
#
#              delay:  Based on the per-tuple and per-page delay for
#                        each node in the query plan
#
#              staleness, accuracy: ignored
#
# Recursively descends the plan tree, keeping track of the number of pages
# and number of tuples generated, and adding up the cost and delay until
# the root is reached.  At this point, the total cost and total delay have
# been calculated.  Multiplies cost by the current load average.  Ignores
# staleness and accuracy.
###########################################################################

# Global variables
set BID 1
set REFUSETOBID 0

#-------------------------------------------------------------------------
# LoadAverage: Utility routine
#
#       Input:  void
#       Output: 5-, 30-, and 60-second load averages
#-------------------------------------------------------------------------
proc LoadAverage {} {
    set result [exec uptime]
    set len [llength $result]
    set result [lrange $result [expr "$len - 3"] [expr "$len - 1"]]
    regsub -all , $result "" res2
    return $res2
}
```

```
#-----------------------------------------------------------------------
#
# CombineBids
#
#      Input:  two bids, bid1 and bid2
#
#      Output: bid that results from combining bid1 and bid2:
#
#              response:  BID if both bid1 and bid2 responses are BID
#                         REFUSETOBID otherwise
#
#              cost:     bid1.cost + bid2.cost
#
#              delay:   bid1.delay + bid2.delay
#
#              staleness: MAX(bid1.staleness, bid2.staleness)
#
#              accuracy:  MIN(bid1.accuracy, bid2.accuracy)
#
#-----------------------------------------------------------------------
proc CombineBids {bid1 bid2} {

    global BID REFUSETOBID

    set response1 [lindex $bid1 0]
    set response2 [lindex $bid2 0]
    set cost1 [lindex $bid1 1]
    set cost2 [lindex $bid2 1]
    set delay1 [lindex $bid1 2]
    set delay2 [lindex $bid2 2]
    set stale1 [lindex $bid1 3]
    set stale2 [lindex $bid2 3]
    set acc1 [lindex $bid1 4]
    set acc2 [lindex $bid2 4]

    set response [expr ($response1 && $response2) ? $BID : $REFUSETOBID]
    set cost [expr $cost1 + $cost2]
    set delay [expr $delay1 + $delay2]
    set stale [expr ($stale1 > $stale2) ? $stale1 : $stale2]
    set acc [expr ($acc1 < $acc2) ? $acc1 : $acc2]

    return [list $response $cost $delay $stale $acc]
}
```

```
#----------------------------------------------------------------------
#
# MERGEJOIN
#
#      Input:  left sub-tree, right sub-tree
#
#      Output: bid
#
# Updates nTuples and nPages - guesses one match for each outer
# tuple.
#----------------------------------------------------------------------
proc MERGEJOIN {nodeNum leftTree rightTree {junk {}} } {

    global BID REFUSETOBID
    global nTuples
    global nPages
    global rtable
    global hostid

    set perTupleCharge .001
    set perTupleDelay .000400

    set leftSubBid [CostBasedBid $leftTree]
    set leftTuples $nTuples
    set leftPages $nPages

    set rightSubBid [CostBasedBid $rightTree]
    set rightTuples $nTuples
    set rightPages $nPages

    # Fill in arbitrary values if nothing is known about
    # the results of the join’s children.
    if {$leftTuples == 0} {
        set leftTuples 10000
    }
    if {$rightTuples == 0} {
        set rightTuples 10000
    }
    if {$leftPages == 0} {
        set leftPages 100
    }
    if {$rightPages == 0} {
        set rightPages 100
    }

    # Each outer and inner tuple is touched once.
    set delay [expr ($leftTuples + $rightTuples) * $perTupleDelay]
    set cost  [expr ($leftTuples + $rightTuples) * $perTupleCharge]

    # Wild guess - one match for each outer tuple
    set nTuples $leftTuples

    set bid [CombineBids $leftSubBid $rightSubBid]
    set bid [CombineBids $bid [list $BID $cost $delay 0.0 0.0]]

    return $bid
}
```

```
#----------------------------------------------------------------------------
#
# NESTEDLOOP
#
#       Input:  left sub-tree, right sub-tree
#
#       Output: bid
#
# Updates nTuples and nPages - guesses one match for each outer
# tuple.
#----------------------------------------------------------------------------
proc NESTEDLOOP {nodeNum leftTree rightTree {junk {}} } {

    global BID REFUSETOBID
    global nTuples
    global nPages
    global rtable
    global hostid

    set perTupleCharge .001
    set perTupleDelay .000400

    set leftSubBid [CostBasedBid $leftTree]
    set leftTuples $nTuples
    set leftPages $nPages

    set rightSubBid [CostBasedBid $rightTree]
    set rightTuples $nTuples
    set rightPages $nPages

    # Each inner tuple is touched once per outer tuple.
    set delay [expr ($leftTuples * $rightTuples) * $perTupleDelay]
    set cost  [expr ($leftTuples * $rightTuples) * $perTupleCharge]

    # Wild guess - one match for each outer tuple
    set nTuples $leftTuples

    set bid [CombineBids $leftSubBid $rightSubBid]
    set bid [CombineBids $bid [list $BID $cost $delay 0.0 0.0 ]]

    return $bid
}
```

```
#-----------------------------------------------------------------------------
#  SEQSCAN
#      Input:  scanIndex, fragIndex, left sub-tree
#      Output: bid
#
# Updates nTuples and nPages based on information in range table.
#
#-----------------------------------------------------------------------------
proc SEQSCAN {nodeNum scanIndex fragIndex {leftTree {}} } {

    global BID REFUSETOBID
    global contract
    global nTuples
    global nPages
    global rtable
    global hostid

    # no extra charge per tuple
    set perTupleCharge 0

    # 5 cents per page
    set perPageCharge .05

    # delay in seconds per tuple retrieved (not including disk I/O)
    set perTupleDelay .000600

    # delay in seconds per disk page accessed
    set perPageDelay .002200

    # Scan on a temporary relation, the result of a sort,
    # join, etc.  Just use the values of nTuples and nPages
    # generated so far.
    if {$scanIndex == -1} {
        set nTuples 10000
        set nPages 100
        set cost [expr $nTuples * $perTupleCharge + $nPages * $perPageCharge]
        set delay [expr $nTuples * $perTupleDelay + $nPages * $perPageDelay]
        set bid [CombineBids "$BID $cost $delay 0.0 0.0" [CostBasedBid $leftTree]]
        return $bid
    } else {

        # Scan on a base relation - set nTuples and nPages
        # from rtable information.
        # Only bid if the fragment is stored at this site.

        set rte [lindex $rtable $scanIndex]
        set frags [lindex $rte 3]
        set fInfo [lindex $frags $fragIndex]

        # Determine if one of the storage sites is this one.
        set storageSites [lindex $fInfo 4]

        set local false

        foreach site $storageSites {
            set storageHost [lindex $site 2]
            if {$storageHost == $hostid} {
                set local true
                break
            }
        }
```

```
        set nTuples [lindex $fInfo 3]
        set nPages [lindex $fInfo 2]

        # If sequential scan is over a fragment that we own, bid on it.
        # Otherwise, subcontract out the sequential scan to another site.
        if {$local} {
            set response $BID
            set cost [expr $nTuples * $perTupleCharge + $nPages * $perPageCharge]
            set delay [expr $nTuples * $perTupleDelay + $nPages * $perPageDelay]
        } else {
            set subPlan "{SEQSCAN $nodeNum $scanIndex $fragIndex}"
            set subBid [subcontract $subPlan $contract]
            set response [lindex $subBid 0]
            set cost [lindex $subBid 1]
            set delay [lindex $subBid 2]
        }
    }
    return [list $response $cost $delay 0.0 0.0]
}

#-----------------------------------------------------------------------------
#
# SORT
#
#       Input:  left subtree
#       Output: bid
#
# Charges a fixed price per tuple and per page.
#-----------------------------------------------------------------------------
proc SORT {nodeNum {leftTree {}} {junk {}} {junk2 {}} } {

    global BID REFUSETOBID

    global nTuples
    global nPages
    global rtable
    global hostid

    set perTupleCharge .001
    set perTupleDelay .000400
    set leftSubBid [CostBasedBid $leftTree]
    set leftTuples $nTuples

    if {$leftTuples == 0} {
        set leftTuples 10000
        set nTuples 10000
    }

    set cost [expr $leftTuples * $perTupleCharge]
    set delay [expr $leftTuples * $perTupleDelay]
    set bid "$BID $cost $delay 0.0 0.0"
    set bid [CombineBids $leftSubBid $bid]

    return $bid
}
```

```
#-----------------------------------------------------------------------------
#
# MERGE
#
#       Input: nodeNum, children nodes
#       Output: bid
#
# Charges a fixed price per tuple for the merge.
#
#-----------------------------------------------------------------------------
proc MERGE {nodeNum subTreeList {junk {}} {junk2 {}} } {

    global BID REFUSETOBID
    global nTuples
    global nPages
    global rtable
    global hostid

    set perTupleCharge .001
    set perTupleDelay .000400

    set bid [list $BID 0.0 0.0 0.0 0.0]
    set mergeTuples 0
    set mergePages 0

    # For each child node, get the bid for the subplan and
    # combine it with the current bid.  Keep track of the
    # number of tuples and pages in the children nodes.
    foreach subPlan $subTreeList {
        set bid [CombineBids $bid [CostBasedBid $subPlan]]
        incr mergeTuples $nTuples
        incr mergePages $nPages
    }
    set cost [expr $mergeTuples * $perTupleCharge]
    set delay [expr $mergeTuples * $perTupleDelay]
    set bid [CombineBids $bid [list $BID $cost $delay 0.0 0.0]]

    return $bid
}

#-----------------------------------------------------------------------------
#
# XIN
#
#       Input: nodeNum, leftTree
#       Output: bid
#
# Charges a fixed price per tuple
#-----------------------------------------------------------------------------
proc XIN {nodeNum leftTree {junk {}}  {junk2 {}} } {

    global BID REFUSETOBID
    global nTuples

    set perTupleCharge .001
    set perTupleDelay .000400

    set leftSubBid [CostBasedBid $leftTree]
    set leftTuples $nTuples

    if {$leftTuples == 0} {
```

```
        set leftTuples 10000
        set nTuples 10000
    }

    set cost [expr $leftTuples * $perTupleCharge]
    set delay [expr $leftTuples * $perTupleDelay]

    set bid "$BID $cost $delay 0.0 0.0"
    set bid [CombineBids $leftSubBid $bid]

    return $bid
}

#--------------------------------------------------------------------------
#
# AGG
#
#       Input: nodeNum, left subtree
#       Output: bid
#--------------------------------------------------------------------------
proc AGG {nodeNum leftTree {junk {}}  {junk2 {}} } {

    global BID REFUSETOBID
    global nTuples

    set perTupleCharge .001
    set perTupleDelay .000400

    set leftSubBid [CostBasedBid $leftTree]
    set leftTuples $nTuples

    if {$leftTuples == 0} {
        set leftTuples 10000
        set nTuples 10000
    }

    set cost [expr $leftTuples * $perTupleCharge]
    set delay [expr $leftTuples * $perTupleDelay]
    set bid "$BID $cost $delay 0.0 0.0"
    set bid [CombineBids $leftSubBid $bid]

    return $bid
}

#--------------------------------------------------------------------------
#
# GROUPBY
#
#       Input: nodeNum, left subtree
#       Output: bid
#--------------------------------------------------------------------------
proc GROUPBY {nodeNum leftTree {junk {}}  {junk2 {}} } {

    global BID REFUSETOBID
    global nTuples

    set perTupleCharge .001
    set perTupleDelay .000400

    set leftSubBid [CostBasedBid $leftTree]
    set leftTuples $nTuples
```

```
    if {$leftTuples == 0} {
        set leftTuples 10000
        set nTuples 10000
    }

    set cost [expr $leftTuples * $perTupleCharge]
    set delay [expr $leftTuples * $perTupleDelay]
    set bid "$BID $cost $delay 0.0 0.0"
    set bid [CombineBids $leftSubBid $bid]
    return $bid
}

#---------------------------------------------------------------------------
#
# UNKNOWN
#
# Don’t bid on plans that contain nodes we can’t identify.
#
#---------------------------------------------------------------------------
proc UNKNOWN {nodeNum leftTree rightTree {junk {}} } {

    global BID REFUSETOBID

    return [list $REFUSETOBID 0 0 0 0]

}

#---------------------------------------------------------------------------
#
# CostBasedBid
#
#       Input:  query plan
#
#       Output: bid
#
# Main procedure.  Looks at token representing the node type and calls
# the appropriate bidding routine.
#
#---------------------------------------------------------------------------
proc CostBasedBid {plan} {
    global rtable
    global hostid
    global contract
    global nTuples
    global nPages
    global BID REFUSETOBID

    if {$plan != ""} {
        set nodeType [lindex $plan 0]
        set bid [$nodeType [lindex $plan 1 ] [lindex $plan 2 ] [lindex $plan 3 ]
                [lindex $plan 4]]
    } else {
        set bid [list $BID 0 0 0 0]
    }

    return $bid
}
```

```
#-----------------------------------------------------------------------
#
# GetQueryBid
#
#       Input:  query plan
#       Output: bid
#
# Main procedure.  Calls CostBasedBid and multiplies result by current
# load average
#-----------------------------------------------------------------------
proc GetQueryBid {plan} {
    global rtable
    global hostid
    global contract
    global nTuples
    global nPages
    global BID REFUSETOBID

    set bid [CostBasedBid $plan]
    set las [exec uptime]
    set len [llength $las]
    set result [lrange $las [expr "$len - 3"] [expr "$len - 1"]]
    regsub -all , $las "" las
    set la [lindex $las 2]
    set cost [lindex $bid 1]
    set cost [expr "$cost * (1 + $la)"]
    set bid [lreplace $bid 1 1 $cost]

    return $bid
}
```