PREDATOR
DESIGN AND IMPLEMENTATION

PRAVEEN SESHADRI

COMPUTER SCIENCE DEPARTMENT

CORNELL UNIVERSITY

TABLE OF CONTENTS

Table of Contents 1

Introduction 4

System Overview 5

Logical Design *5*

Code Design *6*

Client-Server Mechanism 8

Server Functionality *8*

Client Connection *8*

Transaction Semantics *9*

Threads package *9*

Client-Server Communication *9*

Overview of Enhanced Data Types 11

Motivation for E-ADTs *11*

Type System Design *11*

Types versus Values *12*

Meta-Information: Specializing Data Types *12*

Query Processing Mechanisms *13*

Indexing Mechanisms *13*

Special-Cased Types *13*

Type Hierarchy *13*

Utilities: Expressions and Values 15

The Expression Hierarchy *15*

UnknownValue *17*

Function Expressions *17*

Aggregate Expressions *17*

Cast Expressions *18*

Expression Plans *18*

Utilities: Records and Schemas 19

Attributes *19*

Record Schemas *19*

Record Structures *19*

Value Arrays *20*

UTILITIES: Shore Storage Manager 21

Architecture Overview *21*

Storage Structures *22*

Miscellaneous *22*

Supporting Relations 23

Relations: Storage 25

Catalogs *25*

Devices *26*

Relations *27*

Flaws in the Design *27*

Relations: Query Language 28

Semantic Checking *28*

Dag Construction *29*

Type Checking *29*

Query Rewrite *29*

Relations: Query Optimization 31

Statistics Collection *31*

Optimization Architecture *32*

Mechanics of Plan Generation *33*

Relations: Query Execution 34

Join Algorithms *35*

Aggregation and Ordering Algorithms *35*

Building E-ADTs 38

E-ADT Interface *38*

Sample E-ADTs *39*

Extensibility: Adding a new E-ADT *39*

Engines: Spanning Data Types 41

Miscellaneous 42

INTRODUCTION

A next-generation database system must support complex data of different types (like images, video, audio, documents, geographic and geometric entities, etc.). Each data type has its own query features, and data repositories hold combinations of different types of data. The broad research focus of the PREDATOR project at Cornell has been to design and build a database system within which various kinds of data can be uniformly and extensibly supported. However, it is a full-fledged DBMS and can also serve as a vehicle for research into other areas of database systems.

PREDATOR is a client-server database system. A major theme in PREDATOR is extensibility of the system --- adding the ability to process new kinds of data. The efficient integration of the different data types leads to interesting design, optimization and implementation issues. Two characteristics of complex data types are crucial: (1) they are expensive to store and retrieve and they are associated with expensive query operations (2) they have rich domain semantics leading to opportunities for automatic query optimization. The primary technical contribution in PREDATOR is the notion of Enhanced Abstract Data Types (E-ADTs). An E-ADT "enhances" the notion of a database ADT by exposing the semantics of the data type to the database system. These semantics are primarily used for efficient query processing, although they serve other purposes too.

This is the first release of PREDATOR. We hope that the system will provide a common platform for database researchers who wish to experiment with new query processing algorithms. This is the reason for providing documented source code with the system. We would welcome and actively help with any extensions to the code that users may provide. PREDATOR implements many features found in commercial relational and object-relational database systems. The emphasis has been primarily on query processing, although transaction processing is also supported. The beta code release includes the ability to execute a large subset of SQL, multiple join algorithms, storage management over large data volumes, indexing, a cost-based query optimizer, and a variety of object-relational features. Basic concurrency control and low-level storage management is handled using the Shore storage manager. A WWW-based graphical user interface allows any Java-enabled Web browser to act as a database client.

The development of PREDATOR has taken more than two years, beginning at the University of Wisconsin, and then growing significantly over the last year at Cornell. Many students have worked long hours and late nights adding code to the system --- this is definitely a group effort. Specifically, Mark Paskin has worked on the project for more than a year and has contributed to many of the code components. In addition, a number of people and organizations helped in various ways with the development of the ideas and the code. While an extensive list is in the Acknowledgements section of the project web site, we especially appreciate the support of Raghu Ramakrishnan, Miron Livny, and David DeWitt at Wisconsin, and the members of the Shore project.

This document is divided into a number of sections. Each section deals with a logical concept and also describes implementation-level details. This is really a developer's design manual, and the discussion of issues can be detailed. In several parts of the document, we have tried to indicate the names of source files that implement the functionality described. We also suggest that the reader look at material (specifically, the class hierarchy) from the project web page to get a better understanding of the code and the overall system.

SYSTEM OVERVIEW

PREDATOR is a multi-user, client-server database system. The system is built using C++, and makes use of inheritance and encapsulation. It uses the Shore storage manager (a database management library developed at Wisconsin) as the underlying data repository. The goal of extensibility is pursued at many levels of the code, but mainly in the type, language and query processing systems. There are two basic components in the system: one is an extensible table of Enhanced Abstract Data Types (E-ADTs) which defines the kinds of data that the system can manipulate. The other component is an extensible table of query processing engines (QPEs). Each QPE provides a declarative query language that can be used to express a query. We shall discuss QPEs in detail in a later section --- at this stage, a QPE should be treated as a module that provides query processing capabilities across one or more data types. For example, much of the relational database functionality is provided by the relational QPE.

In its beta release, PREDATOR runs on Solaris 2.5 running on Sparc platforms. In about 65,000 lines of C++ code, it implements a relatively lightweight, but highly functional OR-DBMS. The system depends on the Shore libraries, and other software packages like Bison and Flex. However, these are easy to install and are not directly supported by the PREDATOR group.

LOGICAL DESIGN

The overall architecture of PREDATOR is illustrated in Figure 1. The server is built on top of some common utilities. These include code to manipulate boolean and arithmetic expressions, simple record and schema classes, and interfaces to the Shore storage manager. There is also code to accept client connections and support multi-threaded execution.

The server has a table of well known E-ADTs, each of which represents a simple type like an integer, a complex user-defined type, or a collection type like a relation or a sequence. The types are essentially independent of each other. There is also a table of query processing engines, each of which defines a query processing language. The various ``engines" are part of the same server; however, each engine does not know about the existence of the other engines. The system is therefore structured as a loosely coupled collection of types and engines. Relations are treated as just another data type, and relational queries are implemented by the relational QPE with SQL as the query processing language. This loose coupling is the architectural mechanism by which the extensibility of types and languages is supported. The desired degree of modularity is almost extreme --- for instance, if any data type were removed, the rest of the system should continue to work. In fact, even if the relation E-ADT and the relational QPE were removed, the remaining system should be functional.

The database server is multi-threaded with at least one thread per client request. Each client communicates with the server using any of the available query languages provided by the QPEs – currently, SQL is the only supported language. Various kinds of clients have been built with different interfaces. Interface functionality is incorporated into the client, and not in the server.

Currently, a client with a textual interface has been implemented. It takes command line options like the server host name and port number. It also has a customizable prompt, a history mechanism and tab completion of filenames (thanks to Gnu readline). More frills and a GUI could be added as desired. It has a few simple commands that it understands; "_shell" executes a shell command, "_consult" redirects input from a file. All other commands are directly passed through to the server, prefixed by the language name.

CODE DESIGN

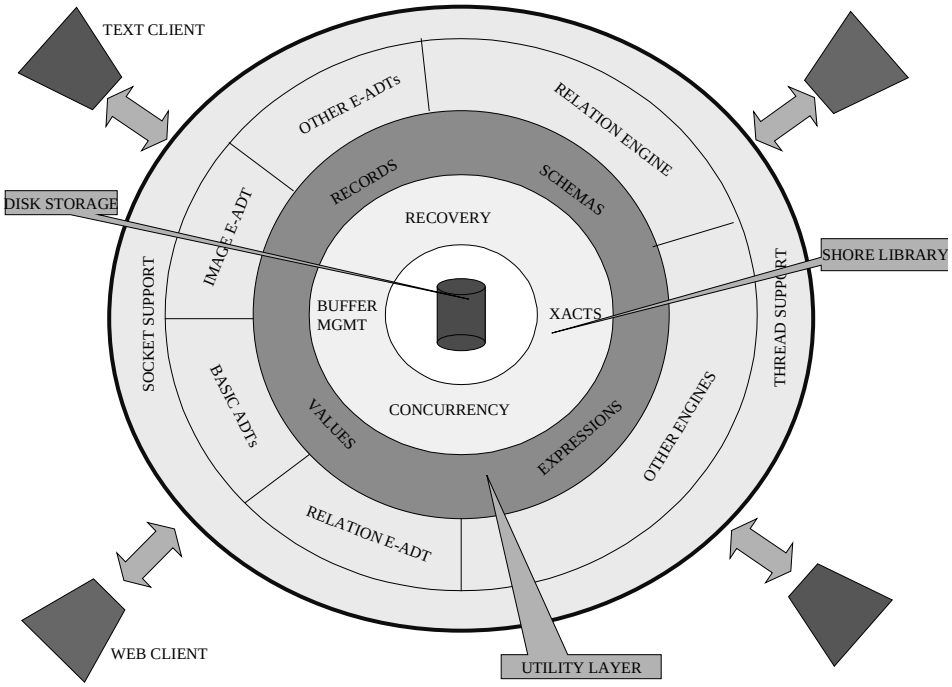

Figure 1 : Server Architecture

The code closely follows the logical design. Each engine and each type is a separate C++ class. In order to promote extensibility, most portions of the code are table-driven. Multi-threading requires that there be almost no global variables, and access to those that do exist should be guarded using mutexes. PREDATOR uses almost no global variables except for the type and engine tables (these are not modified after server startup). Shared structures within these (like structures within an engine) should always be accessed via a mutex.

The Shore storage manager uses its own threads package to implement non-preemptive multi-threading. The reasoning behind the use of a non-preemptive scheduler is that database activities will block when they go to the disk for data access, or when they need to acquire a lock for synchronization. These points serve as "punctuation marks" in the thread execution, at which the thread voluntarily relinquishes its control.

Some rules have been followed (at least in spirit) in the code design:

- Follow coding conventions: Detailed coding conventions have been specified, though not followed as closely as would have been preferable. Many global class names and variables have an "Xxx" prefix. This does not stand for anything in particular, but the letters Xxx are easy to search and replace with any suitable prefix. This way, we do not expect naming conflicts when PREDATOR is extended with new libraries. Most of the classes dealing with relations are prefixed with "Rel".

- Maintain modularity: Care was taken to keep each module, each type and each engine independent. Further, since all interactions are through uniform interfaces, modularity and extensibility should result.

The code layout is as follows. There are separate sub-directories with different pieces of functionality. Each contains both .C and .h files. The directories of interest are:

- server : Contains the core code for threads, error handling, types and engines.

- client : Simple client providing a basic SQL text query interface.

- socket : Library with calls for socket use.

- utils : A collection of utilities that can be used by all E-ADTs and engines. This includes a variety of expressions and values, records, schemas, abstract query trees, and importantly, the Shore storage manager.

- newtypes : Code for handling new data types, and some of the simpler new types.

- reladt : Several layers of subdirectories for a major data type – relations.

- web : Java client which is meant to run within a browser like Netscape or IE.

CLIENT-SERVER MECHANISM

PREDATOR is a client-server system. There are many possible flavors of client-server systems. One important decision is between a multi-threaded server and a multi-process server. The PREDATOR server is multi-threaded --- when a client connects to the server, a new thread is created to service the client's requests. The conventional wisdom is that this is more efficient than a multi-process server. In our case, the designers of Shore, who have provided their own threads package, made the design decision for us (but more about that in a bit). Our client-server mechanism is straightforward, but we have tried to structure it in an extensible fashion to support a variety of clients of different complexities.

SERVER FUNCTIONALITY

The server starts up with an initialization thread which performs system startup (*server/xxx_thread.[Ch]*). The initialization thread has three tasks ---

(1) It creates an XxxEnv environment structure (*server/env.[Ch]*) which is responsible for reading command-line parameters, and setting up the execution environment for the server. Important components of the environment include the input, output and error streams. Another important component is the error stack, which is useful for tracing error messages across function calls.

(2) It then creates a server thread and a monitor thread. The monitor thread is provided to act as a ``DBA window''. Eventually, this will allow facilities like registration of valid users, their passwords, etc. Currently, it only allows listing of connected clients and some simple system statistics. This is open to improvements. The server thread waits for connections from clients.

(3) It waits for completion of the server and monitor threads and then exits.

CLIENT CONNECTION

The server thread waits on a well-known socket port. For each client connection, a request thread is created to service a client session. Each such thread "sees" an image of the whole PREDATOR server (this is the standard way threads work, of course). The server maintains a client structure (XxxClientInfo) for every connected client (*server/client.[Ch]*). The associated information is the user name, the user id, the total execution time for this client, and any other statistics that need to be maintained. The client thread has a handle on this client structure. The thread itself contains an XxxEnv environment structure (a collection of defaults for each client) and an error stack (used to return error messages). It is important that at various places in the code, all of this information can be accessed (for instance, defaults that affect the behavior of the system vary from client to client). Fortunately, the current active thread object is always globally accessible (via a macro call XXX_THIS_THREAD( )). From the thread object, one can access the desired information.

The XxxEnv structure carries all the "global" information that each thread needs. Importantly, it allows each E-ADT and each QPE to create state information for each client. For instance, an E-ADT may choose to create a temporary file to hold the intermediate results for each client. This is associated with the XxxEnv and is reclaimed when the XxxEnv is deleted (when the client closes the connection).

TRANSACTION SEMANTICS

There are two ways for a client to execute transactions. One approach is using the "trans begin", "trans commit" and "trans abort" commands. This allows the client to package a bunch of SQL statements as a single transaction. An easier mechanism is to treat each client input as an individual transaction (see *server/client.C*). This is the default mechanism used. Shore provides the underlying transaction mechanism, so this part of the design is simple.

THREADS PACKAGE

It is important that the threads package used in a database system be compatible with low-level operations like buffer management, concurrency control, etc. Consequently, threads in PREDATOR are based on the threads package provided with Shore. Scheduling of threads happens in a non-preemptive fashion; a thread blocks only on I/O, and that is when thread switches can occur. Each thread has an associated priority, and within each priority group, the scheduling is round robin. We currently do not make use of thread priorities. Each thread is provided with a pre-allocated 64K stack space (which is a compile time parameter that can be modified). The benefit of this is that thread switches are fast. The negative is that the stack should not grow arbitrarily (meaning, no recursive deletes of lists, and no static large arrays). Another negative is that this limits the number of threads that can exist in the system at the same time.

All I/O operations in Shore are executed via the threads package, so that when an I/O request blocks a thread, another ready thread can execute. The package also allows threads to read from pipes or from stdin without blocking. It allows for non-blocking reads and writes on socket connections. More details are provided in a later section on the Shore storage manager.

In retrospect, I believe that the decision of the Shore group to roll their own threads package was an unfortunate one, especially since they used a non-preemptive threads package. If thread switches only occur at known places, the code tends to have less stringent concurrency control safeguards. Consequently, it is now rather difficult to port Shore code to use a different preemptive threads package (like the native Windows NT threads). Such native packages are likely to be optimized for high performance. Further, Shore threads only yield when they perform I/O. While this is fine for traditional database operations, I am doubtful that this is the right approach for object-relational applications which have compute-intensive functions. However, it is easy to criticize in retrospect; at the time that development started on Shore, there were few standard threads packages that worked across hardware platforms.

CLIENT-SERVER COMMUNICATION

Currently, all communication from the client to the server is textual. When the client connects to the server, there is an initial protocol-establishment phase, which involves some setup parameters being exchanged. While there is a nice framework to support a variety of protocols (*server/protocol.[Ch]*), the existing protocols are pretty simple. Eventually, our hope is that these protocols can be extended to support industry-standard interfaces like ODBC. After the initial setup, the client sends queries over the socket in the form of ASCII text, and receives replies. The current client-server protocol requires that the client specify the language in which the query is phrased (since the server can support many languages). For example, if the language is SQL, the client might send across a query string of the form "SQL: SELECT * FROM Emp". The prefix serves as an indicator to the server of the query language.

At the server end, the client's thread treats the input socket as a file, and reads from it using the thread-safe file_read_hndl mechanism. It uses the prefix to invoke the parser from the appropriate query processing engine on the file. The output and error streams in the client's XxxEnv environment are also aliases for the socket to the client. Both of these are treated as ordinary C++ output streams, so most of the code simply uses stream I/O to send data or error messages back to the client. It is not correct, however, to simply create a standard output stream from the socket connection. Consider the possibility that the client connection is extremely slow. The results being written from the server may fill the socket buffer, causing the *entire server to block*! Instead, any writes to the socket must go through the thread-safe mechanism provided by the Shore threads package. By the time this became an issue, we already had several thousand lines of code with methods that would print to a C++ "ostream" class. Changing all of that to use file_write_hndl was out of the question. Instead, we define our own ostream (XxxSockStream) class that incorporates the thread-safe writes. This is a subclass of ostream, to which we pass a C++ streambuf. This streambuf is a subclass of the standard C++ streambuf, with the overflow method overloaded. You can think of this as simply a character buffer, which acts as the stream. When the buffer fills up, the iostream library code automatically calls the overflow () method to clear up some space in the buffer. Our overloaded method writes out the contents of the buffer to the output socket at this stage, using the file_write_hndl mechanism. If the socket buffer is full, only this one thread will block, not the whole server (see *utils/sockstream.[Ch]*).

We have implemented two basic kinds of clients. One is a simple ASCII text-based client (*client/txtclient.[Ch]*) whose protocol requires the server to write back the answers in ASCII form. Therefore, this client is nothing but a shell that sends queries, reads the answers and displays them. The other client is a Java applet that operates in a Web environment. While this is explained in detail later, one important distinction is that the server writes out raw binary data in response to queries. The Java client handles the actual formatting of the results.

OVERVIEW OF ENHANCED DATA TYPES

An Enhanced Abstract Data Type (E-ADT) is the basic abstraction used to support complex data in PREDATOR. In current object-relational systems like Postgres, Paradise, Illustra, Informix, etc, ADTs are used to encapsulate the behavior of new data types. This makes the system extensible --- new ADTs can be added or deleted independently. A naïve summary of the features of standard ADTs is as follows:

- they specify the storage format for values
- they specify methods (usually read-only) that can be invoked on values
- they specify how some methods can be matched using indexes

As an aside, we should mention that this discussion is limited to "value-ADTs" --- i.e. we are not trying to also define an object model with inheritance, identity, etc. There is little consensus in the database community about a desirable object model – in fact, this is probably because object models are so powerful. Empirical market evidence suggests however, that the greatest attraction of object-relational systems lies in the ability to encapsulate data using value-ADTs. There do not seem to be killer database applications on the horizon that really need inheritance and identity – the OO-DBMS products adequately service such niche applications. Given this background, we were glad to bypass the bloody "object-orientation" battlegrounds and focus on encapsulation using value-ADTs.

MOTIVATION FOR E-ADTS

Encapsulation and extensibility are obvious desirable properties, and we have no desire to mess with them. However, we also care about efficiency --- which seems to have been mostly ignored in previous OR-DBMS systems. We are dealing with complex data types, whose methods can be very expensive. Often, the cost of a query execution is dominated by the execution of methods, not by joins or traditional database I/O. Surely, it makes sense to try to optimize the methods themselves and the combinations of methods. For example, if the expression *Data.image.sharpen().clip()* occurs in an SQL query, it probably is more efficient to treat it as *Data.image.clip().sharpen()*. This is the basic motivation for E-ADTs --- to enhance the data type interface so that the semantics of methods can be used to improve efficiency. The improvements that result are often of several orders of magnitude. It turns out that it is non-trivial to support cost-based or statistics-based optimizations. The main issue is that the statistics need to be maintained on an E-ADT specific basis, and propagated throughout the code along with type information.

Extending the same idea, why not allow an E-ADT to provide semantics for other operations, not just query optimization? Why not allow each data type to support its own query language, multiple evaluation mechanisms for each method, specialized indexing techniques, etc? While the PREDATOR framework does support many such extensions, our current focus has been on query optimizations.

TYPE SYSTEM DESIGN

All E-ADTs are subclasses of XxxADTClass (*server/types.[Ch]*), which has a large number (approximately 20) of virtual methods. A number of useful subclasses of it have been defined (*newtypes/genadt.[Ch]*) so that new data types can be easily defined. The type system design is at the heart

of the system architecture, and so a good understanding of its features is necessary. We emphasize the following aspects:

- Distinction between type objects and values belonging to the type
- Meaning and use of meta-information
- Representation of values within data records
- Definition of methods of the data type
- Definition of aggregate methods of the data type
- Query processing and optimization interfaces
- Statistics collection mechanisms
- Indexing mechanisms

In a later section, we will use an example to demonstrate how a new data type should be built. At this stage, a high-level description of the design concepts is provided.

TYPES VERSUS VALUES

Each XxxADTClass (or subclass) defines the behavior of the data type ----- the actual values of the data type are not treated as instances of the class. For instance, the image E-ADT is defined by XxxImageADT (*image/image.[Ch]*), but actual images are not treated as instances of XxxImageADT. In fact, there is exactly one instance of each E-ADT class created, and all these instances are placed in the type table (actually, the XxxADTArray indexed by the type identifier).

A record is the basic storage abstraction. A relational tuple is mapped to a single record. Every E-ADT needs to specify how its values will be stored within a record as an attribute. Both fixed-size and variable-size formats are supported. For large data types, it makes sense to store the actual values as a separate object and store the handle to that object in the record. It is the responsibility of the E-ADT to transform the in-record representation to some appropriate in-memory representation before using it.

META-INFORMATION: SPECIALIZING DATA TYPES

Every data type in PREDATOR can be associated with "meta-information". This meta-info is used to specialize the data type. As an example, consider an image data type. A particular column of a table may be of this type, but with the following special characteristics: it is limited to JPEG images, and it needs to be stored at two additional lower resolutions. This information is specified as part of the schema of the relation in the CREATE TABLE statement. Some current systems create one new data type for each such variant of a well-known type. Obviously, there are an infinite number of possible data types, if a new type needs to be created for each such. Instead, the approach used in PREDATOR is that there is only one image type; however, it is specialized by the meta-info that is associated with each attribute.

The term "meta-information" is sometimes used to indicate information maintained on a per-object or per-value basis. In our case, the meta-info is maintained on a per-attribute basis rather than a per-value basis. The same meta-info mechanism is also used to maintain statistics on a per-attribute basis. These statistics as well as the other meta-info can be used while optimizing the execution of methods of data types.

QUERY PROCESSING MECHANISMS

Every E-ADT can define methods that operate on its values. For example, the Image data type can define a method *Sharpen ()* that sharpens any edges in the image. This method can be used in an SQL query. For example, SELECT C.Photo.Sharpen() from Cars C. The *Sharpen()* method is **not** a member function of the Image E-ADT C++ class. Rather, the C++ class provides some generic interfaces that provide type checking, optimization and execution of all the methods of the E-ADT. The interface for optimization is important --- every expression in an SQL query is "optimized". In the case of an expression involving a method invocation, the corresponding E-ADT is asked to optimize the method. This mechanism allows each E-ADT to apply arbitrary optimizations to expressions involving its methods.

INDEXING MECHANISMS

Each E-ADT should be able to specify its own indexing structures, and should be able to adapt existing indexing structures to its special characteristics. This component of PREDATOR has not yet been implemented. Currently, two kinds of indexing structures (B+ and R* trees) are supported. All indexes must belong to one of these categories (as the section on Indexing describes, more indexing structures can be added). B+ trees require the key values to be integers, doubles, or strings. For R* trees, the key values must belong to E-ADTs that provide a bounding box (using the GetBoundingBox() method). The current implementation of R* trees in Shore supports 2-D bounding boxes, and consequently, that limitation is carried over to PREDATOR.

SPECIAL-CASED TYPES

While the ADT mechanism provides extensibility, it can also be inefficient for commonly used types like integers. We would like PREDATOR to compare favorably with standard relational systems on traditional queries. If every operation on an integer is going to require a member function invocation, the system will perform significantly worse than current systems. This problem occurs because the ADT idea is carried to an extreme --- we might as well special-case a few standard data types and make the rest available through an extensible table. This is the approach in PREDATOR. We have treated integers (XXX_INT), doubles (XXX_DOUBLE) and strings (XXX_STRING) in a preferential fashion. This takes the form of switch statements in the code --- something common in non-extensible systems. The plan is that all the SQL-92 types could be special-cased in this fashion. We haven't really concentrated on supporting all those types however, but we could if it proved to be a priority. The special-cased types also have XxxADTClass objects representing them in the XxxADTArray; however, much of the functionality of the types is hard-coded rather than accessed through the class.

TYPE HIERARCHY

The XxxADTClass in *server/types.h* has more than 20 virtual member functions. It sounds like it may be complicated to define new types. To make it easier, several subclasses of XxxADTClass have been defined. One is a XxxDummyADTClass, which is used for the special-cased types, Integer and Double.

There is also a XxxStringADTClass for variable length strings. Two primary decisions need to be made with respect to the storage of a new E-ADT:

- Is the actual data for an attribute of this type going to be stored within the record? If so, the E-ADT is considered to be "Small". Otherwise, it is "Large", and a handle to it is stored within the record.

- Is the in-record storage requirement for the E-ADT (either the whole value for Small types or the handle for Large types) of fixed size? Variable-size values are stored at the end of the record with the fixed-size slot holding an offset to the actual data.

Based on the answers to these two questions, four generic categories of ADTs are defined in *newtypes/genadt.[Ch*]. Most new data types will be defined as subclasses of one of these categories.

- XxxSmallFixedADT: This is the simplest kind of E-ADT. Use this for simple structures (XY points, complex numbers, etc.)

- XxxSmallVariableADT: This is suitable for variable-length structures like strings, filenames, etc.

- XxxLargeFixedADT: Most complex ADTs fall in this category. The handles to the actual data objects usually contain the OID and any additional extracted feature information.

- XxxLargeVariableADT: This category is useful for complex ADTs where some of the extracted information may include variable-length strings.

The developer of a new data type can choose to subclass the appropriate ADT class, depending on the desired behavior and properties. Whatever the choice, there are various virtual member functions that usually need to be implemented. Later in the document, we have examples of how a data type can be built.

UTILITIES: EXPRESSIONS AND VALUES

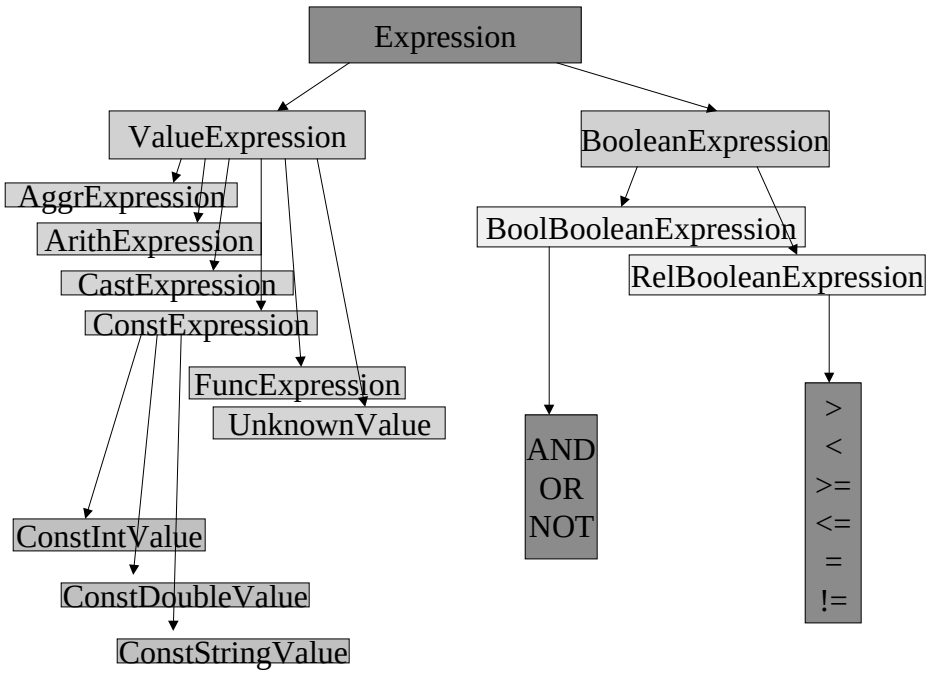

Figure 2: The Expression Hierarchy

The DBMS needs to interpret different kinds of expressions and values --- starting from simple integers and arithmetic, and building up ones that are more complex. PREDATOR uses a hierarchy of expression classes. The root of the hierarchy is the XxxExpression class (*utils/expr.h*) which specifies some standard interfaces that all expressions support. XxxExpressions are compile-time data structures ----- they are created when queries are parsed, and they are used for type checking. When the query is optimized, each of the expressions within it is also optimized; specifically, the Optimize() method of the XxxExpression class must be called. This returns an instance of (a subclass of) XxxExprPlan (*utils/exprplan.h*). It is the expression plan that is used to evaluate the expression. This separation between compile-time structure and run-time structure is a basic design decision. Because we always Optimize() every expression, this provides the opportunity to optimize any nested sub-expressions that might need to be planned (remember, that is the whole point of E-ADTs). Of course, one might say, why go through this hassle for an expression like 1+2 ? One reason is that the expression could have been 1 + f(2), which might need optimization. A more mundane reason is that this separation leads to much cleaner code.

THE EXPRESSION HIERARCHY

XxxExpression provides for the possibility that an expression can have one or more "children" or sub-expressions. The important aspects of its interface are the routines that help with type checking.

Remember that expressions occur most commonly in SQL queries, where there can be more than one table in the FROM clause. When an expression references a column of a table (eg. A.field1 > 5), the column reference needs to be "resolved". The system has to determine that it corresponds to a particular field of a particular table of the FROM clause. In order to do this, the relational query processing engine uses the ResolveReferences() method of XxxExpression. This method is invoked on each expression once for each table of the FROM clause. Once the references have been resolved, the column references (called XxxUnknownValues in PREDATOR ---- see *utils/values.h*) can determine their types and the expression's type can be checked by calling TypeCheck(). After TypeCheck () is called on the value expression, its return type as well as the return meta-info can be repeatedly accessed through member functions.

All expressions are either boolean expressions (XxxBooleanExpression) or value expressions (XxxValueExpression). Booleans expressions usually occur only in a WHERE or HAVING clause. Value expressions are all other expressions and they must all have a return type. Booleans are not a valid data type in PREDATOR ---- a field of a table cannot be declared of type boolean. Hence this distinction between boolean and value expressions. In retrospect, this was an awful and shortsighted design decision ---- actually made by oversight, rather than consciously. It would be convenient to allow user-defined functions to return boolean values, and use these directly in a WHERE clause ---- unfortunately, we cannot do this currently. While it is not a big change conceptually to eliminate this distinction, it is too much hassle for now. However, we plan to make this change eventually.

XxxBooleanExpressions have two sub-classes. The BoolBooleanExpressions are formed using AND, OR, or NOT. The RelBooleanExpressions involve all the arithmetic comparisons (>, >=, <, <=, =, <>), and equality on any data type. The sub-expressions of a BoolBooleanExpression are also boolean expressions, whereas a RelBooleanExpression has value expressions as children (see utils/expr.[Ch])

XxxValueExpression has several subclasses listed here:

- XxxUnknownValue (*utils/values.[Ch]*) ------ reference to a column of a table
- XxxArithExpression (*utils/values.[Ch]*) ------ any kind of arithmetic expression
- XxxConstValue (*utils/values.[Ch]*) ---- any constant value
- XxxConstIntValue
- XxxConstDoubleValue
- XxxConstStringValue
- XxxFuncExpression (*utils/funcexpr.[Ch]*) ----- a function or E-ADT method
- XxxAggrExpression (*utils/aggrexpr.[Ch]*) ----- an aggregate expression
- XxxCastExpression (*utils/castexpr.[Ch]*) ----- explicit cast of an expression to a new type

It is important to know the specific subclass of a value expression in some situations. For example, in a SELECT clause, some of the expressions may be aggregate expressions, in which case they need to be treated differently. It would be nice to be able to directly check on the class of an object ---- but

C++ does not allow this (other languages like Java do). Instead, each value expression contains an enumerated ValType variable that identifies the subclass to which it belongs.

UNKNOWNVALUE

These are column references, usually of the form <tablename>.<fieldname>. Though SQL allows the tablename to be omitted in some cases where it is implicit, the implementation of SQL in PREDATOR almost always requires the tablename. An option is to use the syntax <tablename>.#<fieldposition>. When the XxxUnknownValue is created, it does not know if the table has such a field, or its type. When asked to ResolveReferences with a specific table, the schema of the table is traversed to check if the tablename matches and if there is a field of the appropriate type. This can be dicey, since the table may be a derived one, and the fieldname may be inherited from a lower level. This is hairy code, and it repeatedly confuses me, although I wrote it. The net result is that if a match occurs, the XxxUnknownValue records SrcChild (the index of the FROM clause table passed into ResolveReferences) and the SrcChildAttrIndex (the index of the field in the schema of the table). What's more, the table schema provides the type and meta-info.

FUNCTION EXPRESSIONS

Function expressions take the form $f(x, y) or x.f(y), the second being a method syntax. Both are represented as XxxFuncExpressions. A function expression has three parts: a function name, argument list, and an optional owner. The owner is an XxxValueExpression. The arguments are a list of XxxValueExpressions (an XxxValueExprList). In order to check if the types match up correctly, TypeCheck() must be called separately on the owner and on the arguments. After this, there is still the issue of checking if the function is a valid one. If there is an owner, its E-ADT registers all valid methods. Therefore, the FuncTypeCheck () member function of the owner E-ADT is invoked. If this is a global function of the form $f(x, y), it belongs by default to the XxxGenFuncADT (see *utils/genfunc.[Ch]*). Very few functions are of this form though.

Every function expression also keeps an XxxFuncParse (see *server/generic.h*) object, which may be null. Since this confuses most people first working with PREDATOR, it bears some explanation. The purpose of XxxFuncParse is to hold any parsing data structure that the E-ADT may have created. This may not make a lot of sense for simple functions. However, consider a complex function --- something like $SQL("SELECT * from Foo") which is a function that executes its argument as an SQL statement (btw, this function is indeed supported in PREDATOR --- read the section on query processing engines). The result of type checking this function includes the parse tree formed by parsing the SQL query.

AGGREGATE EXPRESSIONS

An XxxAggrExpression is created from a SELECT or HAVING clause when a functional syntax is observed; e.g. max(A.x). Actually, since there could be a parsing conflict with global functions (of the form f(x,y)), the global functions are prefixed with a dollar sign within an SQL query. So in a query "SELECT foo(x) ….", foo(x) is an aggregate expression, whereas in "SELECT $bar(x) …", bar(x) is a function expression. We have the restriction that an aggregate has no more than one argument. I'm not sure why we have this restriction, though it makes it slightly simpler to have extensible aggregates (read on below). Every aggregate expression is associated with an XxxAggrOp, which encapsulates the behavior of the aggregate function. There are some well-known aggregate functions (min, max, count, etc.), each of which has an XxxAggrOp. In PREDATOR, each E-ADT can define its own aggregates as well. All such are represented by an ADTAggrOp. All the standard XxxValueExpression interfaces like TypeCheck() are channeled through the ADTAggrOp to the E-ADT of the argument to the aggregate.

E-ADTs have a member function AggrTypeCheck () which verifies that the aggregate expression is valid.

CAST EXPRESSIONS

Any expression can be cast to a different type using the standard typecast syntax (e.g. "SELECT [int](1.0) …."). PREDATOR does not perform any automatic typecasting except in simple arithmetic expressions; this is possibly something we should add in a later release. An XxxCastExpression has a child expression, which is the source expression that is being cast. It also has the desired cast type --- this destination E-ADT must decide if the cast is allowed. E-ADTs have a CastCheck () member function that performs this task. However, this does break the modularity of E-ADTs, since one E-ADT must know about another in order to validate the type check. This is not as bad as it appears because the destination E-ADT does not need to know the details of the implementation of the other E-ADT.

Every data type has a default cast --- from a quoted string. In order to execute this cast, the ReadText () method of the E-ADT is invoked (see the section on E-ADTs).

EXPRESSION PLANS

Each of the XxxExpression classes defines a subclass of XxxExprPlan. The primary interface that is provided is the Evaluate () member function, which is used to execute the expression. Most of the different expression plans are simple, and usually contain all the information of the corresponding expressions. The XxxFuncPlan has an XxxFuncPlanInfo structure that serves a similar purpose to the XxxFuncParseInfo, except that it is part of the plan, not part of the expression.

UTILITIES: RECORDS AND SCHEMAS

Record structures (*XxxRecord*) and record schemas (*XxxRecordSchema*) are provided as basic utilities for all data types (*utils/record.[Ch]*). A record schema is logically an array of attributes, each of which defines a named field and its type. A record is an array of bytes representing an instance of the schema. The record can only be interpreted in the context of the schema.

Records are designed to move seamlessly from disk to memory and vice-versa; the structure of the record does not change. In some systems, when a record is brought into memory, every field is "swizzled" into an in-memory format, leading to the creation of an in-memory record structure. When an in-memory record needs to be written to disk, the in-memory records are converted to a disk-storable form. In PREDATOR, the responsibility for swizzling is left to each E-ADT to perform in a lazy fashion.

ATTRIBUTES

An attribute (XxxAttribute) consists of a field name, a type identifier (index into the XxxADTArray) and the meta-info for values of this attribute. There are also a couple of extra fields used for type checking attributes involved in queries. These attributes should not be part of the XxxAttribute structure, since they have no meaning on disk --- expect a change in this part of the structure soon.

RECORD SCHEMAS

Each schema (XxxRecordSchema) has an array of attributes, along with the length of the array. There is also additional information indicating which of the attributes forms a key. Once the schema is constructed with the attribute information, it is possible to compute the offsets at which each attribute will be found within a record of that schema. The ComputeStats() method accomplishes this --- every new record schema created must have this method called on it before it used to interpret a record.

RECORD STRUCTURES

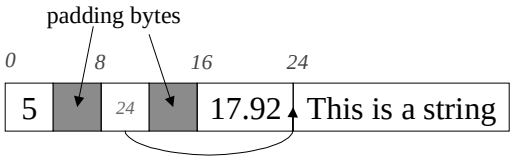

Figure 2: The record <5, "This is a string", 17.92>

An XxxRecord has two components: a size in bytes and a data area, which is a contiguous byte array. This is a main-memory structure, but by design, the main-memory structure is very similar to the structure of a record on disk. The entire contiguous byte array is stored as a single object on disk, and when necessary, is brought into memory as a record. There are three issues to consider in the design of record structures: (a) how are variable-length fields handled? (b) how are null values handled? (c) how is field alignment handled?.

The first portion of the record contains one fixed-size region per field. The size of each region is determined by the corresponding E-ADT (which is known from the schema). For example, an integer is 4 bytes, a double is 8 bytes, etc. Variable-length fields like strings use a 4 byte "offset" --- the actual data is stored at the end of the record, and the fixed-size slot holds the integer offset within the record to the start of the variable-size data. We do not currently handle NULL values, but when we do introduce them, they will be handled using a bitmap at the start of the record. Finally, every field is aligned on an 8-byte word boundary. This is especially important for complex E-ADT fields. As the next section discusses, the Shore storage manager guarantees that objects are returned aligned on 8-byte word boundaries. If within each record object, we align each field as well, we are guaranteed not to have alignment problems. In fact, I do not think that we need to perform any memcpy()s while accessing fields of records! However, some parts of the code currently do have memcpy()s in them – we will need to go through them and determine if they are really needed.

I'd like to emphasize the importance of the XxxRecordSchema in dealing with records. Without a schema, a record is just a byte array. The schema provides all the information needed to access the fields of the record. In fact, much of this information is pre-computed (in ComputeStats()), so that the offset of any field can be quickly determined. The E-ADTs also play an important role in the record structure. Each E-ADT provides a member (StoreInField) to specify whether the field should be placed in its fixed-size slot or at the end of the record. For some E-ADTs, the size of the field can depend on the meta-information associated with the field. For example, there is a "bytearray" E-ADT, which as the name suggests, is a fixed-size array of bytes. The size of the array is specified as part of the meta-information for this field. In our current implementation, the bytearray is stored outside the record as a separate object. However, if it chose to be stored within the object, the size of the field would depend on the meta-information indicating the array width.

VALUE ARRAYS

An XxxValuesArray is a useful abstraction that helps with the construction of records. In order to create a record, we need the schema and all the attribute values. The record cannot be constructed incrementally since it has to be allocated as a contiguous byte array (and this requires the entire record length to be known initially). Of course, one could do this using repeated allocation and copy, but that would be inefficient. Instead, an XxxValuesArray is used as an accumulator of record values. It is simply a main-memory record structure without the contiguity. Here's how it is typically used.

Consider an operator like a file-scan, which projects a few fields of its input records. Based on the schema, an XxxValuesArray structure is created. For each input record, the desired output fields are accumulated in the XxxValuesArray, and then an output record is created from it. There is one copy of the values from the array to the record. The flexibility gained is definitely worth the extra overhead.

UTILITIES: SHORE STORAGE MANAGER

The Shore storage manager (SM) is a library whose functions are invoked to support all disk-based storage in PREDATOR. They also provide transactional semantics to disk-based data access and multi-threading. Access to the SM is encapsulated by the XxxStorageManager class (see utils/shoresm.[Ch]). While Shore defines an ss_m class, PREDATOR wraps this within the XxxStorageManager class. Ideally, no part of the code outside of shoresm.[Ch] should even know of the existence of the Shore SM library calls – they should work exclusively through the XxxStorageManager interface. Currently, there are still a few places in the code that break this encapsulation --- we plan to remove them soon. The basic issues we're going to discuss are:

- Architecture overview
- Devices and volumes
- Objects and identifiers
- Files and indexes
- Sorting
- Transactional semantics

The discussion of threads was part of an earlier section.

ARCHITECTURE OVERVIEW

The SM is meant to be part of a database server (or Value Added Server – VAS, in Shore terminology). There is a buffer pool, which maintains a main-memory cache of the disk-resident data. The size of the buffer pool is determined by a parameter to the constructor. It is expected that each transaction running in the server has its own thread of control. If necessary, there may be several threads for the same transaction. As described earlier, the SM implements non-preemptive threads that yield only at a point when I/O needs to be performed. The reasoning is that I/O often causes processing to block, in which time another thread could be run. However, the threads package itself is run within a single UNIX process without using any native threads support. Apparently, it is difficult to perform non-blocking disk I/O from a UNIX process. This necessitates the use of separate processes (called *diskrw*) for I/O, and complicates the design. The buffer pool is placed in shared memory that is accessible by the primary server process and by the diskrw processes. There is one diskrw process per "device" (think of this as one diskrw per logical disk). I/O requests from the server process are sent as request messages to the corresponding diskrw process. The thread on the server now blocks waiting for a response from the diskrw. However, this does not block the entire server process --- other threads may run.

The rest of the architecture is standard. There is log-based recovery based on the ARIES mechanism, and there is support for raw disks or disks simulated on top of UNIX files. We have chosen the latter for development, because it is much easier to deal with. If performance is the primary consideration, we should switch to raw disks.

STORAGE STRUCTURES

The SM deals with data stored in "devices", which correspond to physical or logical disks. While the original Shore design was to allow each device to contain many "volumes", which were logical partitions, the current implementation associates one volume with each device. Each volume has a "root index" which serves as the directory for the volume. Typically, one would create files and indexes in a volume and use the root index to be able to look them up. A file is a collection of objects --- all objects must be created within some file. Every object has an OID that uniquely identifies it. Files and indexes are also special objects, identified by OIDs. An index is an association of key values to data entries, which are usually OIDs.

Objects can grow arbitrarily large (check the Shore documentation for the upper bound on physical object size). There is about 50 bytes of header information associated with each object, and this can constitute a significant space overhead for small objects (like tuples of a few fields). Shore provides the notion of logical OIDs, which use one level of indirection to identify objects. This gives the underlying SM more flexibility in how it moves objects around. Logical OIDs typically consist of a volume ID and an object ID within it. While this is the default suggested by Shore, there is a faster physical OID version of all the SM interfaces that can be used. We will consider moving to this version if the Shore project plans to continue supporting physical OIDs.

The XxxStorageManager encapsulates most of the SM functionality. The file utils/shoresm-defs.h defines XxxShoreVolumeId, XxxShoreObjectId, XxxShoreLogicalId and XxxShorePhysicalId. One global instance of the storage manager, called XxxShoreSM, is created as part of server initialization. The XxxShoreSM always contains one volume and a file within it – these are the default storage containers for objects that do not have their own explicit files and volumes. Many of the XxxStorageManager interfaces provide wrappers around the underlying SM interfaces – for instances, there are several methods to read an object, given its OID.

The SM uses the notion of a "Pin" to read objects. Since objects can be large, they may span several pages. One option would be to pin all the pages of the object in the buffer pool --- but this could be a bad idea for very large objects. Instead, the Pin abstraction provides an iterator over the large object. We have wrapped the Pin abstraction within an ObjectIterator class --- however, there are some parts of the code that haven't migrated to the new abstraction yet.

MISCELLANEOUS

A simple ACID transactional semantics is supported by the SM. The recovery levels can be varied by changing the storage properties of a file. Four storage properties can be specified: regular, append (logs old data but not new ones), load (logs only control structures) and temporary (no logging). Concurrency control can occur at various granularities (page-level, file-level, object-level). We currently do not really utilize the various options. Instead, we stay with the defaults (page-level 2PL with regular files).

The SM also provides a sorting utility that we use extensively to sort files for relational queries.

SUPPORTING RELATIONS

We will examine the relational support in some detail over the next few sections. The code that handles relations resides in the *reladt* subdirectory. There are two components to this support; a relation E-ADT and a relational query-processing engine (QPE). The basic design is that the relation E-ADT provides the standard E-ADT features. Any field of a record can be a relation – in other words, PREDATOR should support nested relations (a.k.a. a complex object model). This feature has not been tested well as the system has evolved, as this has not been the focus of our activity. In the next release, we hope to be able to eliminate any errors that may have been introduced into this part of the code.

The relation E-ADT acts like any other data type, providing all the standard member functions --- in fact, it must behave like any other data type if it is to be used as the type of a single field. The meta-info associated with a relation is the schema of the relation. No member functions are supported. Within a record, a handle to the actual relation is stored. The relation itself is stored as a separate object. The relation E-ADT by itself is not very interesting, and so we will not discuss it particularly.

Most of the complexity is in the relational QPE (XxxRelationEngine). This supports the query language SQL, the creation and storage of relations and catalogs, optimization of SQL queries and execution of these queries. To better understand the distinction between the E-ADT and the QPE, read the section on QPEs. The reladt directory contains the following subdirectories:

- basic --- contains the definition of the relational E-ADT (reladt.[Ch]) and the relational QPE (releng.[Ch])

- storage --- defines the storage and retrieval of relations, the catalogs used to characterize them, and the manipulation of storage volumes

- compile --- parsing and type checking for SQL queries

- optimize ---- query optimization for results of parsing

- execute --- run-time execution of optimized queries

The QPE functionality is implemented in the XxxRelationEngine class with some of the functions encapsulated in separate classes for data storage, optimization, and query execution. The names of these classes are RelDataEngine, RelOptEngine and RelExecEngine respectively. The suffix "Engine" has persisted from an earlier incarnation of the code base, though these are really "sub-engines" of the main relational QPE. Instead of using "Xxx" as the prefix on all the class names, "Rel" is used within the reladt directory.

Each of the four components (storage, compilation, optimization and execution) will be described in detail in its own section. In this section, some high-level issues are covered. The main design decision made was that PREDATOR would not be specialized in anyway for relational queries. As far as I know, every other comparable RDBMS or ORDBMS differs from PREDATOR in this respect. Our design goal was that it should be possible to compile the system and use the system without the reladt directory (and hence without any relational functionality). How does this make sense? If there are any other QPEs provided, they would be usable. We were lead to this design almost by accident because I

started out building a database system to handle time-series data exclusively. Along the way, I wanted to also support relational data and found that my systems design was inadequate. So I threw it away and started again, leading to PREDATOR. One of these days, we will actually try to compile the system without reladt and see what happens.

Any incoming query is marked with the name of the query language it is posed in – SQL queries are prefixed with the string "SQL:". This allows the system to route them to the relational QPE's ExprParse() method. The original design was meant to allow multiple relational QPEs, each with a different query language. For instance, it is conceivable to think of an ORACLE_SQL and a SYBASE_SQL QPE additionally. The XxxRelationEngine includes a RelDataEngine, a RelQueryEngine, a RelOptEngine and a RelExecEngine as members. The query parsing, optimization and execution are totally functions of the relational QPE, so we can think of the RelQueryEngine, RelOptEngine and RelExecEngine classes as being purely simple encapsulations of these pieces of functionality. The RelDataEngine implements all the storage functionality for the relational QPE. However, the relation E-ADT also needs to be aware of it, to implement methods that store and retrieve meta-information. Ideally, the relation E-ADT should be independent of the relational QPE.

RELATIONS: STORAGE

The storage of relations is handled by the RelDataEngine. The design goal was to keep the storage mechanism as extensible as possible, in keeping with the overall system extensibility. Therefore, we started out shooting for multiple repositories, multiple storage mechanisms, multiple representations of a relation, etc. While the design looks nice, at some point we settled for a single storage structure and ignored all the fancy extensibility. Therefore, the design appears more heavyweight than is needed if we had started out shooting for one specific storage structure. The design also has flaws if one cared to use the extensibility features, but I think the flaws can be fixed (more about this in a bit).

The RelDataEngine (see *storage/reldata.[Ch]*) keeps an array of RelDataSrc objects, each representing a source of relational data. Each RelDataSrc can be implemented in any manner as long as the interface is supported. We implemented only one data source however, the RelShoreDataSrc on top of Shore (see *storage/shorerel-src.h*). The RelShoreDataSrc maintains an array of "devices" – these correspond to logical volumes or disks that are managed by Shore. There is the potential for confusion in terminology here --- Shore has the notion of a device (which is analogous to a disk or partitions of a disk) and a separate notion of a volume mounted on a device. Any stored objects are associated with some volume, so a Shore volume is an important abstraction. The current Shore version allows no more than one volume per device, which makes a Shore device and a Shore volume logically equivalent. We encapsulate this common notion in a RelShoreDevice class (see *storage/shorerel-dev.h*). Every RelShoreDevice has a RelShoreCatalog (see *storage/shorerel-cat.h*) which stores information about all the relations on that device. So far, simple enough.

The relational engine also defines the catalog information maintained for each relation. The RelCatalogInfo class (see *storage/relcat.[Ch]*) has three components: XxxRecordSchema, XxxImplInfo and XxxStatsInfo. The RelCatalogInfo is the meta-information of the relation E-ADT. The XxxImplInfo component describes how a particular relation has been implemented. This includes a list of indexes on the relation, and any other information like its sort order, etc. The XxxStatsInfo contains statistics available on the relation, including the cardinality, and any per-column statistics collected. Remember that RelCatalogInfo and all its components are in-memory data structures, but the information in them has to be stored on disk in order to persist across invocations of the database system. Therefore, there is a translation from the RelCatalogInfo to a disk-resident format and vice-versa. When a table is created using a CREATE TABLE command, a RelCatalogInfo structure is created and propagated to disk. At a later stage, when the table is accessed in a query, the RelCatalogInfo for the table is constructed from the stored catalog information.

CATALOGS

The storage of catalogs on disk follows the System-R technique of keeping the information in standard tables.

There are currently five catalog tables in PREDATOR:

- _STABLES (tablename, arity)
- _SATTRS (tablename, attrname, attrindex, attrtype, key?, attrmetainfo)
- _SINDXS (tablename, indexname, indexexpression)

- _SSTATS (tablename, statsinfo)
- _SASTATS (tablename, attrname, attrstatsinfo)

Most of the tables and fields in them are obvious. The attrmetainfo field of _SATTRS is a fixed size field that stores the meta-info of the attribute. This field is defined by the MetaInfoHandle (see *server/types.h*) and is of type XXX_TYPE_BLOB. The idea was to make it simple for data types to define meta-information that could be persistently stored. Most data types have very small amounts of meta-information to be stored. Instead of forcing the E-ADT to define its own storage format, PREDATOR allows the E-ADT to use a fixed-size field, which is just an array of bytes. If the meta-info is too large to fit within a MetaInfoHandle, it is the responsibility of the E-ADT to store it as a separate object (or possibly many objects) and to keep a handle to it in the MetaInfoHandle. The nice part is that this is transparent as far as the relational catalogs are concerned. When constructing a RelCatalogInfo from the stored catalog data, they have to create the XxxAttributes that form the schema. Since the type of each attribute is known, the corresponding E-ADT is asked to interpret the attrmetainfo field and return an appropriate meta-info structure (see CreateMetaInfo() in *server/types.h*). The _SSTATS table maintains per-table statistics while the _SASTATS table keeps per-attribute statistics. The collection of statistics is explained in the section on query optimization, but at this point, it is worth noting that each E-ADT determines what statistics are to be collected and stored. The statsinfo field of _SSTATS and the attrstatsinfo field of _SASTATS are both of type XXX_TYPE_BLOB.

The data stored in the catalog relations describes all the tables in the device, including themselves --- this last is the hairy part. Once a device has been mounted with these relations available, it is easy enough to obtain the catalog information for any relation name --- this is the purpose of the GetCatalogInfo () member function (see *storage/shorerel-cat.C*). It sets up a scan of each table in the catalog to retrieve the pertinent information for the specified relation name. This scan uses the same code that every other scan uses – this is supposed to be one of the advantages of storing catalogs in this fashion. Some indexes are built on these tables and they are used if appropriate.

There are two parts of the code that are complicated. One is the creation or "formatting" of a RelShoreDevice. The different catalog relations have to be created and populated with tuples that hold information about themselves. Similarly, when a device is mounted, the catalog relations need to be accessed in a bootstrapped fashion. The RelShoreCatalog keeps in-memory handles on all the catalog relations (i.e., it creates an XxxRelation object for each of them – see *execute/relation.h*). These in-memory handles allow scans on the data in the relations. The problem is that in order to create the XxxRelations, their RelCatalogInfos are needed, which in turn depend on the catalogs. This bootstrapping can get a little complicated.

DEVICES

Every device needs to be created once – this formats the "volume". This stage also involves the creation of the catalog relations. Later, the device needs to be mounted before the data in it can be used. When mounting the device, a RelShoreCatalog needs to be created. How will it find the catalog relations? The analogy in a file system is, how does one get to the root directory? The answer is simple --- there must be some well-known location on the device. Shore provides such a notion – actually, something called a "root index" on each volume. The RelShoreDevice creates a simple B+-tree index, which is stored into the root index. All the catalog relations are recorded in this index. The extra level of indirection is probably superfluous.

Devices are also responsible for creating an XxxRelation based on its name. This usually happens in a two step process. The first step is to retrieve a handle on a relation (a RelHandle). This usually contains the object id of the Shore file that holds the relation. The next step is to create the XxxRelation object from the handle. There are different variants of the GetRelation () member function that handle these steps.

RELATIONS

Each device should be able to support different implementations of a relation. The default implementation maps each relational tuple to a single Shore object and models the relation as a Shore file (a collection of objects). This implementation is called ShoreFileRelation. We did try a couple of other implementations to test the extensibility, but over the last year, we've settled on ShoreFileRelation. There is some overhead associated with this design. Since every tuple is a separate object, Shore has to assign it an object id, and some additional header information.

FLAWS IN THE DESIGN

Since catalogs are associated with each device, the presence of more than one device is going to lead to duplicate catalog tables! This is an obvious problem with the design. However, this does not affect us currently, since we use only one device.

RELATIONS: QUERY LANGUAGE

This section deals with the parsing of SQL queries in PREDATOR. The parser is constructed using Flex and Bison. The implemented version of SQL is not SQL-92 compliant, though it contains many of the important features. Features not supported include NULL values and integrity constraints. In terms of the design, one important consideration is that parsing of queries should be reentrant. Why? Because we support nested relations, and nested SQL queries. In fact, every stage of query processing (parsing, type checking, optimization and execution) should be totally reentrant. Most standard lexical analyzer and parser tools (like lex and yacc) use global variables, which makes them non-reentrant. Luckily, Flex and Bison provide options that create reentrant scanners and parsers.

The SQL parser (*compile/sql.y*) and scanner (*compile/sql.l*) are rather straightforward. The SQL parser generates a RelViewClass data structure (*compile/relquery.[Ch]*) which has individual fields for the SELECT, FROM, WHERE, GROUPBY, HAVING and ORDERBY clauses. The generation of this structure simply means that the input was syntactically correct --- no semantic checks have been performed. Complex queries with multiple query blocks are parsed into a RelViewClass with nested RelViewClass objects within it. The first step in semantic checking is to convert the RelViewClass structure into a query block structure. The viewToOpDag () function performs this translation into a directed graph of "query blocks" called RelOperators (see *compile/relops.[Ch]*). Actually, the viewToOpDag () function creates a RelOpDagBuilder object, whose task is to construct the graph of RelOperators from the parsed query. Each RelOperator corresponds to one SQL query block or to a base table. The RelOperator graph is usually a tree, except when the use of relational views makes common sub-expressions easily detectable. There is actually a graph of RelQueryNode structures, each of which holds a RelOperator. RelQueryNode is a subclass of AbsQueryNode, an abstract query graph class defined in *utils/absquery.[Ch]*. The root class has routines for the traversal of graph structures. This portion of the code has been inspired by the design of Starburst and DB2 --- specifically the QGM structure, which the RelOperator graph is a poor imitation of.

There are a few subclasses of RelOperator --- RelSPJOp (for select-project-join), RelBaseOp (for base tables), RelGroupByOp (for grouping and having), RelOrderByOp (for order by). The translation from the SQL query blocks to RelOperators isn't exactly one-to-one. We break an aggregate query into a separate RelSPJOp and a RelGroupByOp above it. This again is modeled on the QGM design in DB2. Each query block will eventually become a unit of query optimization, without much interaction across query blocks. Therefore the decision to split the grouping and the joins is limiting --- it prevents us from applying recently proposed techniques that move aggregations ahead of joins and vice-versa.

SEMANTIC CHECKING

The construction of the RelOperator graph (by methods of RelOpDagBuilder) can involve some non-trivial steps. The first step checks that the query is well formed. A query is deemed "semantically incorrect" if any of the following hold:

- It specifies two target expressions with the same name, e.g.,SELECT S.foo1 AS bar, S.foo2 AS bar FROM table S;
- The WHERE (selection) clause has an aggregate expression
- It specifies a grouping without any aggregates
- There is a target expression that is not a grouping or aggregated term, when there is at least one aggregate.
- There is an expressions in the HAVING clause which is neither an aggregate nor a grouping term.

- The same expression appears twice either in the grouping list or ordering list (ORDER BY clause)
- There is an ORDER BY expression which is not a grouping term or aggregate, when there is at least one aggregate.
- The same expression is used in an aggregate and to group on.

Undoubtedly, some cases are missing, and we will add them as they are discovered.

DAG CONSTRUCTION

A query graph is always constructed in the same way. There is always an initial SPJ node, which takes as inputs all of the relational inputs to the block. Then, if aggregates are referenced, a GBY (Group-by) node is placed above. If an ordering is specified, an OBY (Order-by) node is placed above. Then, finally, an optional final SPJ node is placed on top. The final SPJ node is needed when it becomes necessary to add "implied" projections to the initial SPJ (or to perform the group selection). We cannot hand off the user-specified projection list to the initial SPJ node for the following reasons:

- It may contain aggregates, which the SPJ node cannot compute.
- It may not include all of the attributes referenced by higher nodes.
- It may not contain all of the expressions referenced by the OBY node.

Therefore, before the graph is constructed, several changes have to be made to the user-supplied projection list. Here, in order, are the steps followed to make the projection list that will be given to the initial SPJ node:

- The list initially consists of all projections the user requested, with the exception of all aggregate expressions.
- All the attributes of the source relations that are used in aggregate expressions, the grouping list, or the HAVING (group selection) clause are added to the list, if they are not already in it.
- All the ORDER BY expressions are added, if they are not already in the list. This is because the OBY node is not equipped to evaluate expressions. It can only sort based on column positions. Any expressions it references are artificially projected in.

If, after the initial projection list has been formed, it differs from the original target list, a final projection list is formed, corresponding to the target list specified by the user. This removes all implied projections from the results.

TYPE CHECKING

To check that a query is valid, the schemas of the inputs and of all intermediate stages need to be determined. Any XxxUnknownValues need to be resolved, and the validity of all expressions needs to be checked. This is performed using a bottom-up traversal of the query block graph (called PropagateMetaInfoUp – see *compile/relquery.C*). It makes sense to start from the bottom since the base relations at the leaves of the graph have schemas that are stored in the system catalogs. There is also a top-down traversal performed after this (called PropagateMetaInfoDown). This currently does not serve a useful purpose and is provided only for symmetry. The original idea was that there may be some information gathered from the bottom-up traversal that needs to then be pushed back down the tree/graph.

QUERY REWRITE

PREDATOR has a query rewrite engine that transforms the query block graph (see compile/relrewrite.[Ch]). Rewrites are specified as rules with a precondition and an action (each is a C++ function). The query rewrite engine is a simple rule engine that tries to fire any rules registered with the system. The purpose is to apply heuristic optimization techniques that span query blocks. Why

is this needed? Because the traditional cost-based optimization that follows later optimizes each query block almost independently. Rewrite rules should be things like

- Pushing projections across query blocks
- Pushing selections across query blocks
- Merging query blocks
- De-correlation
- Magic sets rewriting
- Eliminating DISTINCT clauses

Each rule is written in a "local" fashion --- it is applied to some specific query block and it can look at and modify that block and its neighborhood. The firing of a rule must move the query graph from one consistent state to another. The rule engine makes two traversals of the query block graph --- one is a top-down traversal and one is a bottom-up traversal. Each rule is registered as part of one of the traversals. We currently have implemented a couple of rules in the released version of the code, though other rules were tried as well. One rule merges two adjacent query blocks (RelSelectMerge), while the other (RelDistinctPushdown) pushes the application of the Distinct property down the query graph. DB2 has a much more elaborate set of rules and a more complex structuring mechanism to order the firing of rules within each traversal.

Eventually, we would like to combine this rule engine with the engine used to fire E-ADT rules.

RELATIONS: QUERY OPTIMIZATION

The code for query optimization is in the reladt/optimize directory. The optimization of a relational query involves the following high-level steps:

- Collecting statistics about the relations
- Enumerating a space of alternative evaluation plans for a query
- Estimating the cost of the plans and picking a plan of low estimated cost

We plan to use PREDATOR as a research vehicle in this area, so the description of this component may change significantly. I should also emphasize that commercial database systems have pretty complicated query optimization code. Our version comes nowhere near it; in fact, even when we release the next version with a fully working dynamic programming optimizer, it will not contain all the special-case code used in commercial optimizers. I consider this a good thing. If there is significant benefit to be gained from an optimization technique, there is likely to be some way to generalize it and implement it in a clean manner. Commercial optimizers, on the other hand, have often added features in response to specific customer queries. The other concern is that optimizers depend on cost estimates, which are usually rather flaky. It is much better to avoid fancy optimizations based on flaky estimates, and stay with tried-and-tested techniques. Having said this, I wish we had the time to implement optimizations like index-only access, detailed histograms, column equivalence, etc. This is fascinating and difficult stuff – and terribly time consuming. It takes a team of experienced developers several months to build a good optimizer, followed by years of tuning and improvements. In PREDATOR, this has been done by a single developer as one among many development tasks, and without really being the focus of our research activity --- so be warned. Over the next few months, our research interest in this component will increase as we look at query optimization issues in dealing with large data types.

STATISTICS COLLECTION

Different statistics need to be maintained on each relation, and on each column of a relation. The original query optimization papers opted for simple statistics (relation cardinality and high-low values for each column) and simplifying statistical assumptions (like uniform value distributions in columns). Recently, there has been a lot of renewed interest in improving the accuracy of query optimization by maintaining more involved statistics (mostly histograms). While we haven't implemented these more recent improvements, they could be added. Statistics are collected by executing a "runstats" command. This is probably something that should only be allowed from the monitor window, but right now, any client can execute it. It is also possible to collect statistics for a single relation by saying "runstats <relationname>". The cardinality of the entire relation is computed by scanning the relation, as are the average tuple sizes. The per-attribute statistics are more interesting. Perhaps a specific category of histograms is ideal for a integer attribute. What are meaningful statistics to collect over an image attribute? Obviously, this should be specified by the image E-ADT. In fact, PREDATOR allows each E-ADT to define its own statistics class (subclass of TypeStats) and to accumulate statistics in it as the runstats scan proceeds. A number of E-ADT methods are provided for this purpose (CreateStatsInfo, AccumStatsInfo, etc. in server/types.h). Statistics collection is almost like an aggregate function --- there is an initialization, an accumulation and a termination phase. These statistics will be passed into the optimization stage when the E_ADT is asked to optimize its methods. As described in the section on relational storage, the per-attribute statistics are stored in the _SASTATS catalog relations. The

relation statistics are stored in _SSTATS. In the current release, the actual storage and retrieval of statistics has not been implemented. We expect to add them along with the full-fledged cost-based optimizer.

OPTIMIZATION ARCHITECTURE

The goal is to construct a query plan from the query specified by the user. Although there is currently no distinction in PREDATOR between query "compile-time" and query "execution-time", commercial systems do allow a query to be optimized once and executed many times. We may support this functionality eventually. So, a query plan is really a compile-time structure – in fact, it is the net result of query compilation. This structure cannot be directly executed or interpreted at execution time. Instead, an execution plan can be generated from it, and this execution plan is then interpreted to evaluate the query.

Like most commercial DBMSs, PREDATOR performs query optimization on a per-query-block basis. The optimization of a query is encapsulated in a RelOptEngine class --- this is simply an encapsulation mechanism. The query block graph is traversed in a bottom-up fashion, and each block is optimized. The optimization of a query results in a query plan, which is a graph (usually a tree) of RelBlockPlan structures (see *optimize/relopt.h*). Each different query block produces its own subclass of RelBlockPlan. The GeneratePlanOp () method of RelBlockPlan creates a RelPlanOp, which is the execution plan that is interpreted for query execution.

Our primary focus has been on the select-project-join (SPJ) query blocks. The optimization of an SPJ block is handled by a RelSPJOptimizer (see optimize/planspj.[Ch]). Actually, there are several sub-classes of RelSPJOptimizer, each with different degrees of complexity.

- Naïve (naivespj.[Ch]) : the join order corresponds to right-to-left order in the FROM clause, and a single N-way nested loop join is created.

- Semi-Naïve Optimizer (seminaivespj.[Ch]) : this varies from Naïve only by creating a left-deep pipeline of two-way joins, instead of a single N-way join. When applicable, a sort-merge join is used instead of nested-loops.

- Greedy (greedyspj.[Ch]) this varies from Semi-Naïve by choosing the join order based on a greedy formula. A not-quite KBZ formula is used, which incorporates costs into the choice of join orders.

- Dynamic (dynspj.[Ch]) this is the System-R style dynamic programming optimizer. The current implementation is not complete, although the enumeration is functional. Missing pieces include interesting orders, correct treatment of indexes, and a clean translation into an execution plan.

The default optimizer is greedy. We plan to add to these optimizer choices as alternatives for research purposes. Currently, various flags can be used to control the choice of tuple nested-loops, page nested-loops and sort-merge joins. Algorithms for order-by, group-by and sort-merge join require that their inputs be sorted. This usually requires the introduction of a sort operation. However, if the data is already sorted in the appropriate order, the extra sort is unnecessary. The proper way to handle this is using the notion of interesting orders in the dynamic programming optimizer. For the other optimizers, we put in a check to make sure that unnecessary sorts are not performed.

MECHANICS OF PLAN GENERATION

Much of the complexity of plan generation arises from logically simple manipulations. For example, is the move from Naïve to Semi-Naïve easy? Logically, the answer is yes, because all we are doing is breaking an N-way join into a pipeline of 2-way joins. However, remember that there are expressions used in the predicates and projections of joins. These expressions include references to columns of tables (i.e. XxxUnknownValues), and these references have been resolved during type checking. A resolved column reference identifies the value as coming from a specific attribute of a specific input relation. However, when intermediate results are thrown in because of the pipeline, all the expressions above it need to be modified to reflect the new positions of attributes. This is also an issue when the join order is being changed. The XxxVarRemap is a data structure used to specify the desired remapping of column references. All expression sub-classes need to define a Remap () method that will apply an XxxVarRemap to itself. We will not explain each of the optimizers here in detail. The code is relatively self-explanatory. Once the dynamic programming optimizer is completed, we will provide a detailed explanation of it. The current implementation follows the classical design with physical and logical properties associated with every partial plan generated.

RELATIONS: QUERY EXECUTION

The code for relational query execution is in the reladt/execute directory. The execution model for queries is similar to the "iterator" model used in many other systems. The basic idea is that a tuple iterator is set up on the highest plan operator, and tuples are retrieved one by one. Each plan operator accesses its children by setting up iterators on them. This leads to a relatively clean and extensible code structure. As long as each plan operator can define an iterator over its tuples, it can coexist with the other operators in the system. While this is not a new idea, some details are important.

From the RelBlockPlan graph created by query optimization, a new graph data structure is created for the execution phase --- this is a graph of RelPlanOp structures. Each corresponds to a "physical" algorithm like a nested-loops join or an aggregation. The RelBlockPlans have a method called GenerateRelPlanOp ()

To avoid confusion, here is a brief recap:

- RelQueryNode and RelOperator are structures that result from parsing a query
- RelBlockPlan structures are the result of optimizing the RelQueryNodes.
- RelPlanOp and XxxRelation are execution-time structures that are generated from the RelBlockPlans.

Of course, there is information copied from one structure to the other; so why did we go with this design? One reason is that it is very clean. All the parse-time structures can be reclaimed after the optimize phase. Likewise, for the other structures. Further, many systems allow a query to be optimized once and executed many times. Although this is not currently supported in PREDATOR, it could be added without much additional effort. The separation of the different data structures is very significant in making this happen. It actually took us quite a few code revisions to finally settle on this design.

Query execution begins in the EvalQuery () method of RelExecEngine (see relexec.[Ch]). The first step is to generate a RelPlanExecutor from the query plan. This is a simple wrapper around the RelPlanOp graph. The purpose of the RelPlanExecutor is to determine what should be done with the results of query execution – should they be stored in a file as for an INSERT INTO command (StorePlanExecutor) , or should they be written to the client, and in what form (PrintPlanExecutor)? EvalQuery simply calls the Execute () method of the plan executor. This has the following stages:

1. Generate an XxxRelation from the RelPlanOp
2. Perform any initialization needed for data output (virtual method of the executor)
3. Set up a scan of the XxxRelation
4. For each tuple, process the tuple (virtual method of the executor)
5. Close the scan
6. Perform any termination needed for data output (virtual method of the executor)

For example, the PrintPlanExecutor may print the result schema as part of initialization, print out each tuple to "process" it, and finally send a termination sequence to the client. The XxxRelation is a main-memory abstraction representing a handle to a collection of tuples. These tuples could be from a stored relation (XxxStoredRelation), or from a computed relation (XxxDerivedRelation). All relations have a scan interface that allows a scan to be initiated, iterated over, and terminated. For stored relations like XxxShoreFileRelation (reladt/storage/shorefile.C), this involves actual access to stored data, using a file scan or an index scan. For derived relations, this requires the use of query processing algorithms. These algorithms are defined as methods of the corresponding RelPlanOp; the methods are InitHandle(), GetNextRecord(), and CloseHandle(). The "handle" refers to the stored state of the algorithm at any stage during the scan. Since the scan needs to produce one tuple at a time, the computation of each algorithm needs to be suspended when a tuple is successfully produced. The computation is resumed on the next invocation of GetNextRecord().

JOIN ALGORITHMS

All join algorithms are provided with the input relations, a target expression list, and four categories of predicate plans (InitPreds, SelPreds, JoinPreds, and ResidPreds). The InitPreds are "initial" predicates that can be applied before any tuple is even retrieved. As a trivial example, the predicate (1 = 2) would fall in this category. So would a nested sub-query that was a constant (except that we don't have these implemented yet). The SelPreds, JoinPreds, and ResidPreds are arrays whose size corresponds to the arity of the join. SelPreds[I] contains selection predicates to be applied to the I'th input. JoinPreds[I] contains join predicates to be applied to the I'th input (i.e. at the I-1'th join). Some join predicates cannot be applied in specific join algorithms. For instance, a sort-merge join only accepts equality predicates. In this case, any extra predicates are specified in ResidPreds[I].

We have currently implemented three join algorithms (i.e. three RelSPJPlanOps); tuple NL join, page NL join and sort-merge join (see spjplanop.[Ch]). The tuple NL join is the simplest, with the only complexity arising because it is an N-way join. The other source of complexity is because the NL join has to be suspended and its state stored in the handle when an output tuple is produced. The page-NL join is more complex because it involves an extra layer of iteration. We also have to be careful with respect to pushing expressions into the scans of inner relations. While join expression can be pushed into inner relation scans in tuple-NL join, it is not correct for page-NL (since there is a whole page of outer tuples, not just a single one). Page-NL is implemented as a sub-class of tuple-NL.

Sort-merge join turns out to be more complicated than we initially expected. As part of InitHandle(), a scan is set up on each of the input relations, and the first tuple is read from each. This seeds the rest of the join algorithm. When GetNextRecord() is called, the entire current equivalence group needs to be fetched from each input. If the two equivalence groups match, then output tuples can be generated. A detailed comment in the code in spjplanop.C provides a further description of the code details.

AGGREGATION AND ORDERING ALGORITHMS

PREDATOR supports aggregation with grouping (see gbyplanop.[Ch]). The generation of groups for aggregation is similar to that for the sort-merge join. Unfortunately, the code is not shared. The current implementation of aggregation and grouping is rather messy, and we hope to clean it up soon. Part of the complexity arises from our desire to support complex E-ADT aggregates, and aggregates that are followed by methods. For instance, we support "*image_avg(image_field).blur(1)*" as well as "*image_avg(image_field.blur(1))*". We plan to further add support for expressions like "*1+max(age)*".

Ordering is needed for the ORDER BY clause of SQL queries. It is implemented using a SortPlanOp (see sortplanop.[Ch]). This uses the sort utilities provided by the Shore SM. The input stream of tuples is written to a temporary file, sorted, and then the tuples are read from the sorted file. Shore also provided a sort-stream abstraction, which is not actively supported; while this should be faster, especially for small sorts, we were unable to get it to work in PREDATOR. It would be a good project to rewrite Shore's sort-stream utility.

RELATIONS: INDEXING

Indexes can be built over expressions on tuples of a single relation. The indexed expression may be something as simple as a single column, or could be arbitrarily complicated. Multiple indexing structures are supported. All of them must be defined as subclasses of XxxIDXClass (see reladt/storage/XxxIndex.[Ch]). There is a global array of supported index structures called XxxIDXArray. The original idea was to place the indexing structures in the utils directory, independent of the relational ADT. However, this did not prove feasible, and we have moved them to the reladt/storage directory. We will probably further reorganize this code in the near future. Each XxxIDXClass defines the behavior of a particular kind of index through virtual methods (just as each XxxADTClass defines the behavior of a particular data type). We currently have two subclasses; ShoreBTree (ShoreBTree.[Ch]) and ShoreRTree (ShoreRTree.[Ch]). The methods fall into the following categories:

- Creation/Deletion of the index
- Inserting/Deleting an index entry
- Scanning the index

Scanning is the non-trivial part of indexing. Typically, some predicate is used to restrict the scan. How does one know if the index matches the predicate? Of course, if the indexed expression is identical to one side of a relational predicate like equality, and the other side is a constant, an exact match results. The Matches() method in storage/relcat.C checks for this. However, there are cases where an index can be used even if the indexed expression is not identical to the expression in the predicate. A good example is with R* trees, where an index on an X-Y point can be used to match an expression *R.point.inside(<constant bounding box>)*. The Match() method of XxxIDXClass is used to provide this information. As a detail, we have to express such a predicate as *(R.point.inside(box) = 1)*, because booleans are not a first-class data type. As discussed earlier, this was a shortsighted design decision, and we hope to change this at some point.

At run-time, the actual scan has to be setup, possibly multiple times if this is the inner relation of an NL join. The Shore index scan interface requires that the range be provided to the scan iterator in a particular format for each indexing structure. New structures may need their own kind of ranges. Therefore, each XxxIDXClass is allowed to define its own IndexRange and its own CursorInfo to be used to store scan state. Methods also need to be provided to merge ranges, since conjunctions could exist with more than one boolean factor matching the same index. Finally, each index structure needs to provide a way to generate a CursorInfo object. This object has methods that allow the caller to perform the index scan iteratively.

BUILDING E-ADTS

In this section, we initially explain the E-ADT interface in some detail. We then present some of the existing E-ADTs and describe how to build a new one.

E-ADT INTERFACE

Each E-ADT has an identifier (which is the index into the XxxADTArray) and a type name. Here are some of the fields of importance:

- *MethArray*: array of method definitions of this E-ADT. Each method definition is a FuncMethodInfo object.
- *AggrMethArray* : array of aggregate method definitions, each of which is an AggrMethodInfo object.
- *MaintainsCatalogs* : is there any meta-information associated with this E-ADT
- *StoreInField* : are the values of fixed-size?

All E-ADTs present an internal interface of about 35 virtual member functions. Most have default implementations, making it easy to define a new E-ADT with just the desired functionality. Here are some of the important methods:

- *IsSmall*() : are the values stored within the record?
- *MaxObjectSize*() : maximum possible size for the in-record representation of a value
- *TypeSize*() : size of a specific object
- *DefaultVal()* : assign a default value
- *TypeCopy()* : make a copy of a value
- *TypeCreate()* : create a new value
- *TypeDestroy()* : reclaim a value
- *OrderedType()* : is there a linear order among the values?
- *Equals()* : equality between two values
- *GreaterThan()*, *LessThan()* : for ordered types
- *ReadText() :* read a value from a text stream.
- *WriteText()* : write out a value into a text stream
- *AddMetaInfo()* : adds meta-information for a specific column
- *GetMetaInfo()* : gets the meta-information for a specific column
- *ExecOIDRequest() :* Execute a callback request.
- *CreateStatsInfo() :* Create an object to collect statistics on this E-ADT.
- *StoreStatsInfo() :* Store statistics information on this E-ADT.
- *RetrieveStatsInfo() :* Retrieve statistics information previously stored.
- *FuncTypeCheck():* Check the type of a method expression.
- *FuncOptimize():* Optimize a method expression.

- *FuncReorganize():* Apply any optimization rules.
- *CastCheck() :* Perform type-checking of a cast expression.
- *CastExec() :* Execute a type-cast.
- *AggrTypeCheck():* Check the type of an aggregate expression.
- *AggrOptimize():* Optimize an aggregate expression.
- *CreateEnv() :* Create an environment(XxxADTEnv) for this E-ADT.
- *DeleteEnv():* Reclaim the environment.
- *GetBoundingBox():* Get a bounding box for a value (used by R* tree indexes)
- *GetCompOp():* Get a comparison operator for matching a function using an index.

SAMPLE E-ADTS

The code release contains several E-ADTs (most in the code/newtypes directory). These E-ADT implementations are provided as templates for other developers to construct other data types. Here is a brief description of each E-ADT : more details are provided on the project web page.

- Audio (audio/*): Audio data in AU format.
- Box (newtypes/box.[Ch]): A 2-D rectangle.
- Bytearray (newtypes/bytearray.[Ch]): A fixed-size uninterpreted array of bytes.
- Circle (newtypes/circle.[Ch]) : A standard geometric type
- Complex number (newtypes/complex.[Ch]): Real and imaginary parts (currently integers – make them floats to really use this!). Good example of methods, etc.
- Counter (newtypes/counter.[Ch]): Automatically incrementing counter values. The current counter is part of the meta-information.
- Date (newtypes/date.[Ch]): Standard date type from the Julian calendar, with a variety of input formats.
- Document (newtypes/document.[Ch]): Simple document type, with HTML, Postscript formats. No methods supported on it yet.
- Image (image/*) : Images in JPEG format.
- Point (newtypes/point.[Ch]) : X-Y point data, useful for geographic applications.
- Polygon (newtypes/polygon.[Ch]) : Multi-vertex convex polygons.
- Raster (newtypes/raster.[Ch]): Specialized type for Sequoia benchmark, representing AVHRR satellite images.

EXTENSIBILITY: ADDING A NEW E-ADT

Currently, the simplest way to add a new data type is to find one of the existing types that is closest in behavior to what you desire, and then modify it. We are in the process of building a graphical E-ADT builder "wizard" that will automate this process. Here are some of the questions you will need to ask:

- Are the values "small" (stored within the record) or "large"?
- Are the values of variable or fixed size?
- Are there methods that will be provided on this E-ADT?
- Are there aggregates provided by the E-ADT?
- Are there optimization rules that this E-ADT will define?

- Is there meta-information associated with this type?
- Will there be statistics maintained on the type?

Once you have answers to these questions, determine which of the existing E-ADTs matches your specification and start from that as a template. If you have all the code for the actual E-ADT methods, it should take no more than a few hours to define most simple types. This is true for all the E-ADTs in the list above, except for audio and image. Once you have defined the new E-ADT class, make sure to register it in server/types.C (see how other types are registered and do the same) and also make sure that it is added to the appropriate Makefile.

ENGINES: SPANNING DATA TYPES

Query processing engines (QPEs) are abstractions that span data types. If each E-ADT is totally modular and independent, where does one place functionality that has to know about multiple types? In QPEs. Here are some examples of such functionality:

- Methods that involve multiple types. Many methods of E-ADTs take parameters of other types. For example, it is comon to have an integer parameter. Even if simple types like integers are special-cased, how about complex types? Methods of Polygon take parameters of type Box. Methods of Video may take an Image as a parameter.
- Cast expressions. Consider an expression that casts a value of type A to type B. In order to process this, types A and B need to know about each other.
- Optimizations spanning types. Currently, E-ADT optimizations are limited by the data type boundaries. If optimizations are to span type boundaries, the modularity of the data types must be compromised.
- Query languages. Any reasonable query language (like SQL) needs to process data of multiple types and specify manipulations of these types.

The point being made is that the data type boundary may not be the suitable unit of modularity for certain pieces of functionality. Currently, QPEs in PREDATOR provide query languages. Cast expressions and methods break the modularity of E-ADTs. Optimizations spanning data types are not currently supported. This functionality will also be migrated to QPEs in the next release.

The two QPEs currently implemented are the XxxDefaultEngine (server/defeng.h) and the XxxRelationEngine(reladt/basic/releng.h). Both are subclasses of XxxDBEngine (server/engines.h), which has the following methods:

- ExprParse() : parses a query and produces a parse structure (XxxGenericParse).
- ExprOptimize() : optimizes the parse structure producing a plan (XxxGenericPlan).
- ExprEvaluate() : executes the plan structure.

There is a global array of QPEs called XxxDBEngineArray, in which all QPEs are entered. Each QPE provides one or more query languages. When a request from a client arrives at the server, the prefix of the request identifies a specific query language. The request is then routed to the ExprParse() method of the appropriate QPE. The relational QPE registers "SQL" as its query language. By default, this creates a global function called $SQL which can be used to construct any function expression (look in server/genfunc.C for the details). The return type is a relation. Likewise, every query language defined by a QPE generates a global function. This allows nested expressions belonging to different languages. An earlier version of PREDATOR had a separate QPE for sequence queries. At the time, SQL queries could have embedded sequence queries, and vice-versa. While the current code release does not have the sequence QPE, the ability to nest language expressions remains. As an interesting special case, an SQL query can have an embedded $SQL() function.

MISCELLANEOUS

Bitmaps

Bitmaps are essential in the construction of a query optimizer. One needs to keep track of the predicates that have been applied and the relations that have been joined into to a partial result. The implementation of XxxBitMap (utils/misc.[Ch]) uses an array of XxxBool (which are chars) to represent a bitmap. Eventually, this implementation should be changed to a more efficient version that uses one bit per logical bit.

Abstract Query trees

Most query languages generate query trees upon parsing a query. Standard operations on these structures include bottom-up and top-down traversals. Instead of repeatedly implementing traversals for each kind of query tree, this functionality is abstracted into the XxxQueryDag (see utils/absquery.[Ch]) class. An XxxQueryDag is a directed graph of XxxQueryNodes. When a relational query is parsed, a RelQueryClass [subclass of XxxQueryDag]object is created, which is a DAG of RelQueryNodes [subclass of XxxQueryNode]. When more QPEs are added, the abstract query trees will be used for their query parse trees as well.

Error mechanism

Whenever an error occurs, the server provides the client with an error stack, if it was compiled in the debugging mode. The error messages provide the file and line number, the nature of the error, and an error message. Almost every non-trivial function or method returns an error code (XxxErrCode) which is either XXX_OK or XXX_NOT_OK. How does one get a return code from a constructor? The correct way to do this is to give every constructor an XxxErrCode& parameter. While we have failed to do this in the current version of the code, any new code should adopt this strategy. Error codes should always be checked. If an error occurs, a line should be added to the error stack using the XXX_NOTE_ERROR(ErrorCode, ErrorMessage) macro. The valid error codes are defined in server/error.[Ch].

There is an extensible error mechanism provided to allow QPEs to define their own error codes. For instance, the relational QPE defines its own error codes (reladt/basic/relerror.h). Errors are noted using the REL_NOTE_ERROR(RelErrorCode, ErrorMessage) macro. The actual error stack entries (see server/error.[Ch]) contain an engine identifier, an error code, an error message, a file name and a line number. The default error codes belong to the XxxDefaultEngine.

Template Classes

A number of useful template classes are define in server/templates.[ch]. In retrospect, these templates should have been used more extensively. Here is a brief list of some of them:

- XxxStack : generic stack with Push(), Pop() and Top()
- XxxMultiSet and XxxSet: collections with iterator scans
- XxxTable and XxxPtrTable : lookup tables of values or pointers to values
- XxxArray and XxxPtrArray : growable arrays of values or pointers to values