PREDATOR: A Resource for Database Research

Praveen Seshadri

Cornell University

http://www.cs.cornell.edu/Info/Projects/PREDATOR/

Abstract

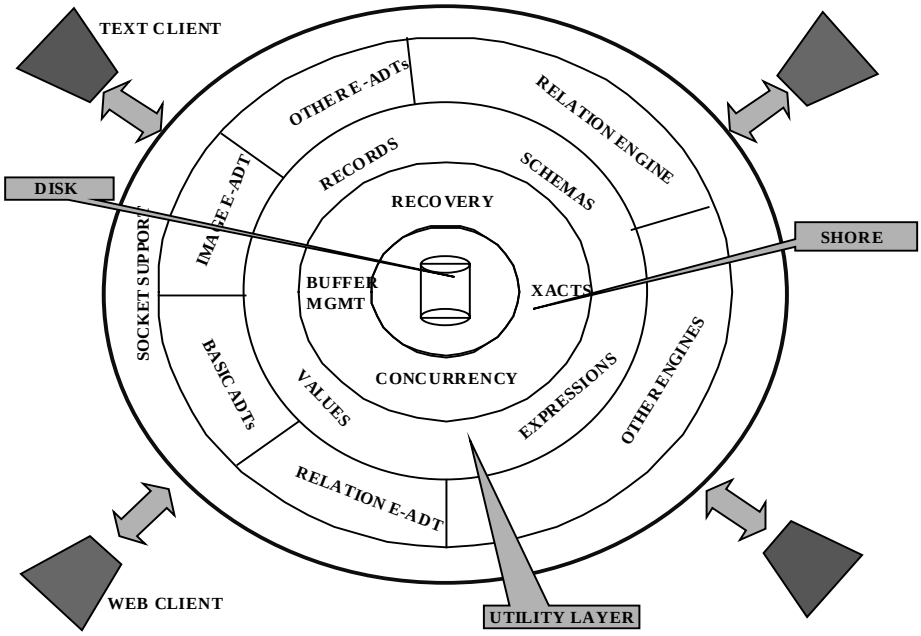
This paper describes PREDATOR, a freely available object-relational database system that has been developed at Cornell University. A major motivation in developing PREDATOR was to create a modern code base that could act as a research vehicle for the database community. Pursuing this goal, this paper briefly describes several features of the system that should make it attractive for database research and education.

Introduction

As a student, I attended a talk by one of my professors who lamented the lack of a common database system that the research community (especially university researchers) could use. Specifically, he compared us to the compiler community that has access to the internals of *gcc,* a comprehensive compiler. In the absence of a common and free database system, university researchers have been left to choose between building standalone prototypes or implementing their ideas within proprietary commercial databases. Some large and well-funded groups can escape this choice by developing full-blown systems, but these are few and far between. Work done in university settings also sometimes suffers in apparent impact since “it was not implemented in a real system”.

I started building the PREDATOR database system when completing my degree. With help from my advisors, the Shore group at Wisconsin, and several students at Cornell, the system is now close to being “real”, and an initial version has been publicly released. Real applications have been built using PREDATOR (including a web-based conference management system used for SIGMOD ‘98. The purpose of this paper is to briefly describe the features available in the system and to promote its use in university research and education. The description of the system is presented in the following format:

1. Architectural overview.
2. Relational query processing.
3. Object-relational query processing.
4. Transaction management.
5. Application development.
6. Current status and future plans.



Architectural Overview

PREDATOR is a client-server object-relational DBMS implemented in C++. The server is a multi-threaded query processing engine built on top of the Shore storage manager [1]. Clients connect to the server using TCP/IP sockets across an intranet or the internet. A client could be implemented in any language (including C++ and Java) and in the case of Java, can run as an applet within a browser. The server creates one thread to process the requests of each connected client. Each thread "sees" an image of the server that is shown schematically in the figure.

The server can be viewed as a collection of Query Processing Engines (QPEs). Each QPE defines a query language and query processing capabilities for that language. The primary QPE that we have built is the SQL QPE, which supports the SQL language with object-relational extensions. However, if one wished to build a specialized multi-media system or a data mining system with its own language, one would model it as a QPE. The QPEs all share the lower layers of the system, primarily the extensible table of data types. Each data type can specify detailed information about its storage format, and optimization techniques for its methods (this is an enhancement over traditional database ADTs, so we call them Enhanced ADTs or E-ADTS)[2]. The behavior of each data type is encapsulated within a single E-ADT class object, and all these E-ADT class objects are stored together in a table. Adding a new type corresponds to adding one entry to this table. The E-ADTs and the QPEs can (and in some cases, should) utilize a number of "utility" services provided through a common layer of the server. Important utilities include storage and transaction management (provided via a wrapper to the Shore storage manager), record and schema management, expression representations, and generic plan representations.

Each client request is tagged with the name of the query language. It is passed to the appropriate QPE, which parses, optimizes, and executes the query. At this high level, it is essentially a vanilla database server architecture, except for the extensible collection of QPEs and the extensible collection of E-ADTs. One important point to note is that the system is not assumed to be primarily a relational database engine. The SQL engine is just another QPE, and is treated on par with the other QPEs.

Relational Query Processing

The relational engine can be described in terms of four components: (a) storage (b) query parsing (c) optimization (d) execution. All these components have various extensibility features built into them. The current functionality is sufficient to run the Wisconsin benchmark and the TPC-D benchmark.

Relational Storage

Relations are stored within logical containers called "devices" that model physical or logical storage devices. Shore provides the capability to map a device to either an OS file or to a raw physical device. Multiple implementations can be provided for a relation. The default implementation is to map each relation to a Shore file, and each tuple to a single object within the file. Shore supports B+-tree and R-tree indexes that associate keys with object identifiers. This facility is used to build relational indexes. Catalog management is handled in the typical fashion through the use of catalog relations. Each relation is described by catalog entries which specify its schema, its implementation information (eg., available indexes) and statistical information for query optimization. It is possible to define new kinds of devices, new relational implementations, and new kinds of indexes.

Query Parsing

PREDATOR supports a version of SQL-92 with object-relational extensions. All the traditional SQL features (SELECT, FROM, WHERE, GROUPBY, HAVING, ORDERBY) are handled. Recently, we have added support for views and nested queries. User-defined functions and aggregates can be embedded anywhere within an SQL query. The parser is not as easily extensible as other parts of the system, in part because of the nature of parser generators like Yacc and Bison (the entire specification of the language grammar is needed in order to generate the parser). However, an important feature of the parser is that it is totally reentrant, which is very important for nested queries and concurrent parsing of multiple queries.

Query Optimization

An SQL query is divided into query blocks, across which query rewrite rules (as suggested in Starburst) are applied. We have implemented a simple query rewrite engine with a few basic rules, importantly a rule to merge SELECT-PROJECT-JOIN (SPJ) blocks where possible. After query rewrite, each query block is separately optimized. For SPJ blocks, several different optimizers have been implemented, some cost-based and some heuristic. A full-scale System-R style optimizer [3] is also available. The cost model combines processor and I/O cost. Simple statistics are maintained for the stored relations. The focus of our work has not been relational query optimization. However, the existing optimizers provide a solid base for others who work in this area. For example, one could easily add code to maintain complex histograms and use them for selectivity estimation.

Query Execution

The execution of a query follows the standard "iterator" model wherein each plan operator asks for the next tuple from its inputs and produces an output tuple. Several join methods have been implemented including different nested-loops joins and sort-merge join. Adding a new join method like a hash-join is simple, since both the optimizer and execution engine use an extensible collection of join methods.

Object-Relational Query Processing

Much of the research component of the project focuses on the support for complex data types (E-ADTs). The basic idea behind E-ADTs is that data types and their methods are not black-boxes. Instead, they can expose some of their semantics, so that the system can optimize queries that uses the E-ADT methods. For example, a query might ask for a small portion of a composite image formed by overlaying several individual images. The naïve execution strategy is to overlay the images and then extract the piece of interest. The image E-ADT might specify that this is equivalent to the cheaper strategy of extracting a small portion from each image before computing the overlay. The details of the E-ADT paradigm and its implications have been described in detail elsewhere [2] -- it is as an important component of our object-relational support. In addition, we support indexes built on functions and function paths, user-defined functions and aggregates written in C++ or Java, and multi-dimensional indexes. As an example of the functionality supported, the Sequoia benchmark [4] over geographic data types (points, polygons and raster images) is easily handled by PREDATOR.

Several data types have been implemented in the system. These range from simple types like complex numbers to large multimedia types like audio and image. DDL statements allow the dynamic creation of new types using alias, enum, and struct constructs. More complex E-ADT definitions may be added through dynamically loaded libraries. The developer of a new data type has a variety of examples to use as prototypes. We also provide a graphical "wizard" that helps with the development of a new E-ADT.

Transaction Management

Every client request is modeled as a transaction and is implemented with full ACID properties. Multi-statement transactions are also supported. PREDATOR utilizes the underlying Shore mechanisms to guarantee transaction semantics. Currently, we have not exploited the various degrees of concurrency and locking granularities that are provided as options in Shore. PREDATOR itself does little to contribute to transaction processing, since most of this functionality is provided by Shore. Researchers who wish to study or modify transaction mechanisms will therefore need to examine Shore code. However, they can measure the effects of their modifications on a complete system.

Application Development

Since our goal was to build a "real" system, application development has been of increasing importance as the system has matured. We have built several web-based database applications that involve queries over collections of multimedia data. One such application was used to handle the submission and review of papers (document data type) for the SIGMOD 98 conference. It was implemented as a collection of CGI scripts that interact with the database server. Similar applications have been built using Java applets as well. A JDBC driver for PREDATOR is also available.

In the introductory database course at Cornell last fall, I asked students to choose a client-server database application and to build it using PREDATOR. This was assigned in the last three weeks of the class. We provided them with sample applications built with CGI and with Java applets. Developing an entire application requires the students to apply classroom concepts in a practical setting.

In upcoming courses, I also plan to use PREDATOR for classroom demonstrations. For example, the query rewrite of nested queries can be visualized through a web-based visualizer. We currently have visualizers for query plans and for E-ADT optimizations as well. Over the next few months, we intend to develop several other visual tools that would prove useful. Once again, this is an area where contributions from others would be welcomed and added to subsequent releases.

Current Status and Future Plans

We released an initial version of the system in August '97. While that version ran only on Sparc/Solaris, our next release, scheduled for April, will run on Windows NT as well. Many of the newer features described in this paper are supported in the upcoming version. While we lack the resources to support ports to other platforms, it is easy for others to perform the ports as long as Shore has been ported to the desired target platform. We are coordinating with the Shore group at Wisconsin to maximize the benefit of a combined storage manager and database system.

Documentation accompanying the initial release includes Javadoc-style web pages defining the classes used in the server and client, suggested sample projects for graduate courses, and a developer's guide that explains the code structure and implementation details. I originally thought that a 50-page developer's guide would be "extensive" but it clearly only scratched the surface. Consequently, our next release will flesh out this document further.

The project is constantly growing -- although research priorities (like conference deadlines) are sometimes at odds with general system development, our schedule is definitely influenced by the needs of users of our system. We encourage members of the database community to use PREDATOR and to give us feedback. Contributions of code and interesting applications would also be extremely useful to add to subsequent releases. More details about the project, the software, and the latest releases are found on the web at: *http:\\www.cs.cornell.edu\Info\Projects\PREDATOR*. And yes, the name is an acronym for the PRedator Enhanced DAta Type Object Relational system.

References

[1] *Shoring up Persistent Objects* : M.J. Carey, et. al. SIGMOD 1994.

[2] *The Case for Enhanced Abstract Data Types* : P. Seshadri, et. al. VLDB 1997.

[3] *Access Path Selection in a Relational Database Management System* : P.G. Selinger, et. al. SIGMOD 1979.

[4] *The Sequoia 2000 Storage Benchmark* : M. Stonebraker, et. al. SIGMOD 1993.