SEMANTIC DATABASE ENGINE DESIGN

Naphtali Rishe, Armando Barreto, Maxim Chekmasov,
Dmitry Vasilevsky, Scott Graham, Sonal Sood
Florida International University, High Performance Database Research center, Miami, FL
Email: {rishen, barretoa , maximc, dvasil01, grahams, ssood001}@fiu.edu

Ouri Wolfson
University of Illinois at Chicago, Chicago, IL,
wolfson@cs.uic.edu

Keywords: Semantic binary data model, database management system.

Abstract: New types of data processing applications are no longer satisfied with the capabilities offered by the relational data model. One example of this phenomenon is the growing use of the Internet as a source of data. The data on the Internet is inherently non-relational. As a result, demand developed for database management systems natively built on advanced data models. The semantic binary data model (Rishe, 1992), satisfies the criteria for the models required for today's applications by providing the ability to build rich schemas with arbitrarily flexible relationships between objects. In this paper, we discuss a new design for a semantic database management system which is based on the semantic binary data model. Our challenge was to design and implement a database engine which, while being native to the model, is reasonably efficient on a wide variety of industrial applications, and which surpasses relational systems in performance and flexibility on those applications that require non-relational modelling. Special attention is given to multi-platform support by the semantic database engine.

1 INTRODUCTION

The Semantic Binary Database Engine is a multi-threaded, multi-platform computer program. Multi-threading allows it to utilize the full CPU power of multi-processor computers. Typically, two different approaches are used for multi-threaded program implementation. One approach is to use one thread per CPU, a queue of work items, and non-blocking operating system calls. Another approach is to use one thread per request with blocking operating system calls. While the first approach allows higher performance, the second approach is easier to implement and requires less effort to port to different platforms.

Multi-platform support allows the engine to be easily portable and to run on different platforms, such as Microsoft Windows, Sun Solaris, HP-UX, and Linux. It makes it possible for a client on one platform to communicate with a server running on a different platform. A detailed discussion of multi-platform support is provided in section 5.

The Semantic Binary Database Engine consists of two major parts – the Database Engine Kernel and the User-Level Engine Environment as shown in Figure 1. The interface between these two parts is the Kernel API, which provides access to the Kernel's functionality.

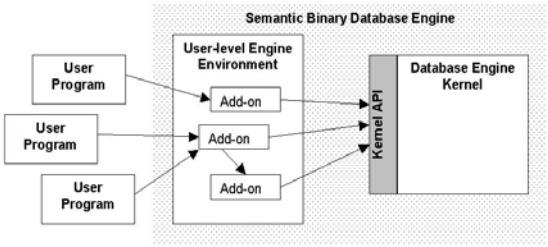

Figure 1: Semantic Binary Database Engine.

The name 'Kernel' does not imply that it runs as a part of the operating system kernel. The Database Kernel consists of tightly coupled modules that provide essential functionality and high performance execution. The User-Level Engine Environment is a set of loosely-coupled modules (add-ons) that have access to the Kernel API and that provide user

programs with the service interfaces. The set of add-on modules may vary depending on the DBMS packaging. Examples of add-on modules are a remote access module and database monitoring tools.

2 DATABASE ENGINE KERNEL API

The Database Engine Kernel API is an interface between the Database Kernel and the User-Level Engine Environment. It has the following properties.
The Kernel API is a set of functions intended to provide the functionality of the semantic database and to hide the implementation details. While the internal implementation of the Database Kernel and modules in the User-Level Environment can be done in an object-oriented fashion, it is preferable to keep the API as a flat set of functions for easier and more efficient inter-process communication.
The interface is accessible, but is not intended to be used by the user application programs. The interface was designed to support efficient execution rather than ease of use. Add-on modules in the User-level Engine Environment provide easy to use interfaces for user programs. This separation provides better reliability and stability than an alternative design with all the software modules fitted into the database kernel since the system can more easily survive faults in modules of the database engine that are running outside the Kernel. It also makes the Kernel code smaller, less prone to errors, and easier to debug and maintain. It is important for the database Kernel not to crash even if some ill-behaved programs misuse the Kernel API.
Since the Database Engine Kernel is a separate process, the Kernel API is an interface between processes running on the same computer. The functionality of remote access is not provided at this level, but is instead provided by the Remote Access Server, which is one of the add-ons in the User-Level Engine Environment. The Remote Access Server can be added or removed from the system depending on the expected functionality of the system. For example, it may not be needed for embedded applications.
The Kernel API handles data in terms of facts that are not yet encoded for any storage structure. This allows users of the interface to see the database in its semantic representation. At the same time, the storage subsystems in the kernel may employ any kind of encoding and data structures to physically store the data.

3 USER-LEVEL ENGINE ENVIRONMENT

The User-Level Engine Environment is a set of modules running as one or several processes on the same computer where the Kernel runs. All these modules (except the first three below) are independent and can be designed and implemented separately. It is important to separate them from the Kernel modules to ensure stability and reliability of the database engine.
The Local Semantic API module provides a conventional semantic API for database applications. This ensures compatibility with old programs that use previous versions of the semantic binary database engine. The semantic API was designed with the assumption that the engine and the user program would run in the same address space (it uses pointers to internal memory structures). While this is faster than remote access, it is not secure since the database engine is not protected from the ill-behaved user programs. This is the reason for the current design to employ a Kernel API to protect internals of the database and to run a Local Semantic API module in the User-level Engine Environment.
The complex query language for semantic databases, called AVDV, is described in (Vaschillo, 2000). The Complex Query Executor analyzes an AVDV query graph, builds the execution plan, then performs queries and obtains results according to the execution plan. Several other components in the User-Level Engine Environment, such as Web Query Tool and Semantic SQL Server require execution of complex queries and rely on this add-on module.
The Semantic SQL Server module allows users to query databases by means of semantic SQL (Rishe, 1999) and standard database protocols such as ODBC. Semantic SQL Server accepts a SQL statement and parses it. Parsing semantic SQL statements involves discovering the relevant sub-tree of the semantic schema given an unambiguous path postfix as described in (Rishe et.al., 2000). The corresponding query AVDV graph can then be constructed. When the AVDV query graph is constructed, it is passed to the Complex Query Executor module for optimization and execution. The result of execution is returned to the user.
The Export/Import module provides export and import of a database into interchangeable standard formats. Some of the common formats are Comma Separated Values (CSV) and Extensible Markup Language (XML). The module also provides export to the proprietary native Semantic Definition

Language (SDL) and to the XML-based Semantic Definition Language (XSDL).

4 DATABASE ENGINE KERNEL

The Database Engine Kernel is the main component of the Database Engine. The Kernel consists of several modules that are tightly integrated to provide maximum efficiency; all the modules run in one address space. The modules have well defined interfaces to communicate with each other, and the internal details of each module's implementation can be designed independently. Figure 2 shows the basic data flow between the modules.

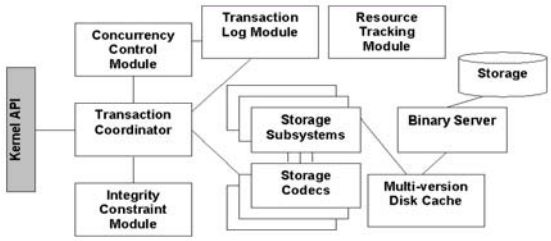

Figure 2: Database Engine Kernel.

The Integrity Constraint Module is used to verify the integrity constraints. The system reports updates to the Integrity Constraint module. For some updates, the system makes an immediate decision that the operation should be rejected. Information about other updates is stored and a decision is deferred until commit time. For example, a decision on a cardinality constraint can be verified right away, while a decision on a totality constraint should be deferred until transaction commit time.

The Transaction Coordinator carries out the transaction lifecycle. The system supports two types of transactions – transactions with versioning and without versioning. The typical execution of a transaction with versioning is shown in Figure 3.

The Concurrency Control module manages transactions and ensures the consistent state of the database. All requests for updates and queries in the system are communicated to the module in the form of lock requests and releases. The Concurrency Control module stores enough information to decide whether to grant or to delay certain lock request. The module also verifies all the locks at the end of a transaction and decides whether the transaction is allowed to commit. Concurrency Control supports several types of transactions and makes the final commit/rollback decision according to the transaction type.

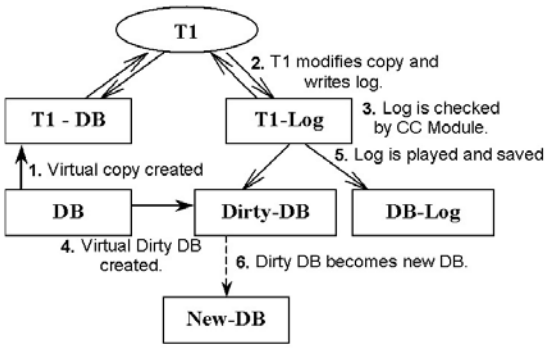

Figure 3: Transaction lifecycle.

All transaction updates are communicated to the Transaction Log module. This module provides storage of the local transaction log on a per-transaction basis. The transaction log can be retrieved later on to be run against the database. This module also maintains the global database transaction log and appends the local transaction log to the global database transaction log at transaction commit time.

Storage Codecs perform encoding and decoding between storage representation and semantic representation. Fact, Record and Bitmap representation can be used in appropriate situations. When the components of the storage item are sent to the module, it composes the storage item that will be placed in the storage subsystem. When the storage item is retrieved from the storage subsystem, the module decodes it to semantic components. For example, the fact representation codec takes two object IDs and a relation ID and combines them into a binary string when the information is to be stored. On retrieval it takes the binary string and parses it into two Object IDs and the relation ID.

The Storage Subsystem is the module which stores/retrieves information to/from files composed of fixed-size blocks. The information is already encoded for storage by the Storage Codec module. A B-Tree structure is the main storage subsystem used in fact and record representation.

The Multi-version Disk Cache improves performance of the disk subsystem by keeping the content of disk blocks in memory and saving disk I/O operations. If a block that is already in the cache is requested, it is not retrieved from the disk and the cached copy is used instead. If subsequent modifications are made to the same block, the cache keeps the block in memory and saves time on disk write operations by waiting until the last

modification is made. In addition to these standard write-back cache functions, the Multi-version Disk Cache provides functionality specific to the database engine. It allows the storage subsystems to lock blocks in memory. Whenever a block is needed by the storage subsystem for certain operations, such as binary search within the block, the storage subsystem does not create its own copy of the block. Instead, it requests the disk cache to lock this block in memory for the duration of operation and uses the same copy of the block. Sharing of the block copy is possible since the modules in the Kernel, including the disk cache, run in the same address space. This type of sharing eliminates the necessity of block copy operations. It is important that the lock is held for only a short period of time, since all the locked blocks have to be present in memory. If locks are held for a long time, the system may run out of memory.

The disk cache provides support for block versioning. A block ID in the cache is two-dimensional: it is composed of a sequential block number in the database file and a sequential database version (B, V). The transaction coordinator increases the database version with every read-write transaction and assigns the version to the transaction. Whenever the transaction requests block B for modification, the block ID is composed of B and the database version V assigned to the transaction. If the block does not already exist, a new copy of the block is created. The new copy is based on the block with the same block number B and the database version that was current at the beginning of the transaction. The old block is retained until a transaction in the system requests it.

The Binary Server module hides the file structure of the database and provides the user with flexible storage options. The module is used to store data of the database engine's files with fixed size blocks to the disk. The Binary Server implements a simple file system that can provide this functionality by using one disk file or several disk files or even raw disks not formatted by the operating system. The Binary Server can also distribute the database across multiple computers. All storage subsystems share the same space of disk blocks provided by the Binary Server. Whenever a block is freed from a storage structure, it goes into the common pool of free blocks. This allows for better management of space allocated to the database.

5 CONCLUSION

Semantic databases have many advantages over relational databases that will allow them to grow in popularity as the complexity of data increases. However, the semantic database engine should be implemented in a way that is not prohibitively expensive on operations typical to relational databases. We have shown that a number of reasonable tradeoffs are possible in the design of the semantic database engine that can make it competitive on applications that are widely-used today.

This work shows how a framework that allows investigation of advantages and disadvantages of different approaches in each of the database engine modules can be built. A number of conclusions have been made on the feasibility of particular choices based on theoretical considerations, as well as practical experience implementing various parts of this design in several combinations.

Innovative technologies missing in the previous semantic database theory and prototype implementations have been designed and discussed. These technologies are expected to overcome some of the shortcomings that have kept semantic databases from being widely accepted in the field.

ACKNOWLEDGEMENTS

This material is based on work supported by the National Science Foundation under Grants No. HRD-0317692, EIA-0320956, EIA-0220562, CNS-0426125, IIS-0326284, CCF-0330342, IIS-0086144, and IIS-0209190.

REFERENCES

Rishe, N., 1992. *Database Design: the semantic modeling approach*, McGraw-Hill. 528 pp.

Rishe, N., 1994. *Semantic Schema Design Language*, available at request, http://hpdrc.cs.fiu.edu .

Vaschillo, A. 2000. *A Semantic Paradigm for Intelligent Data Access*. Ph.D. Dissertation, Florida International University, 143 pp.

Rishe, N., 1999. *Semantic SQL*, available on request at http://hpdrc.cs.fiu.edu .

Rishe, N., et.al. 2000. SemanticAccess: Semantic Interface for Querying Databases. In *Proceeding of the VLDB Conference*, pp. 591-594, September 10-14, 2000, Cairo, Egypt.