SimpleDB: A Simple Java-Based Multiuser System for Teaching Database Internals

Edward Sciore
Boston College
Computer Science Dept
Chestnut Hill, MA
617-552-3928
sciore@bc.edu

ABSTRACT

In this paper we examine the problem of how to give hands-on assignments in a database system internals course. We argue that current approaches are inadequate, either because they are not sufficiently comprehensive or because they require using software that has a steep learning curve. We then describe *SimpleDB*, which is software written expressly for such a course. SimpleDB is a database system in the spirit of Minibase. Unlike Minibase, however, it supports multiple users and transactions via JDBC, and its code is easy to read and modify. We then describe a course that we teach using SimpleDB, and discuss the educational benefits resulting from it.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *relational databases, query processing.*

General Terms

Algorithms, Design, Languages

Keywords

pedagogical database software, database internals, minibase

1. INTRODUCTION

The topics in an undergraduate introductory database course typically fit into two categories: how to use a database system, and how a database system works. Topics in the first category include database design, relational algebra, SQL, and how to build database applications. Topics in the second category include concurrency control, recovery, indexing, and query processing.

Historically, these topics were compressed into a single-semester course. More recently, many schools (including Boston College) have chosen to teach the material over two courses, with each course corresponding to one of the two categories. The first course becomes a user-oriented course, whereas the second course becomes a system-oriented course.

There are several advantages to this division. The first advantage is that it is possible to go much deeper into the various topics. The user-oriented course can cover advanced material such as web-based development and data mining. The system-oriented course can cover additional design alternatives, such as locking vs. multi-version concurrency, top-down vs. bottom-up query processing, additional forms of indexing, etc.

The second advantage to the split is that each course has different prerequisites and a different student population. The user-oriented course requires very little programming experience (if any), and is appropriate for students looking for a practical, applications-oriented course. For example at Boston College, this course is taught in the business school and is a requirement for their Information Systems concentration. Computer Science majors tend to find the course very easy, and it counts towards their major only as a low-level CS elective.

The system-oriented course, on the other hand, is a quintessential computer science course, on a par with the traditional upper-level CS courses. It touches on mainstream issues such as data structures (for indexing), external sorting, operating systems (file systems and memory management), distributed systems (client-server, threading, deadlock), language interpreters, and algorithm design. At Boston College this course has only Data Structures as a prerequisite, but an interesting alternative would be to teach it as a senior capstone course.

Another difference between the two courses is the kind of assignments given to students. The user-oriented course is very hands-on: Students learn to use a particular database system, and very often work on a major project in which they build a sophisticated application from scratch. The system-oriented course, on the other hand, tends to be theoretical: Students draw pictures of B-Trees, interpret log files by hand, prove serializability, and calculate optimal query plans.

A theoretically-based system course is ok, but I think most would agree that a hands-on course is better, at least for undergraduates. In my experience, undergraduate students have a relatively easy time solving problems in a narrow context, but have a very difficult time grasping how everything fits together. Ideally, a student would write an entire database system as part of his coursework, as one would write a compiler in a compiler course. However, database systems are much more complex than compilers, and so that approach is not practical. What should an instructor do?

Several strategies are possible:

- Have students write a "toy" database system from scratch.
- Provide students with the object code to an instructor-written database system, and have students rewrite specific modules according to a given API.
- Provide students with the source code to a pedagogically-written database system, and have students modify specific modules.
- Give students access to an open-source commercial database system, and let them add features to it.

Each of these approaches has drawbacks.

Writing a toy system gives students the experience of building a large system, but this system usually winds up being so simple that it has no correspondence to how a real database system is built.

On the other hand, writing individual modules to an otherwise-unknown system gives students the experience of writing actual code, but does not give them an overall understanding of the system and does not allow them to make improvements that encompass multiple modules.

The last two strategies allow students to see the entire source code of a system. Studying this source code can provide the sense of how things fit together, and modifying the code can reinforce this understanding. The difference between these two strategies is whether the code is specifically written for student use, or whether it is commercial-grade.

Commercial open-source software has the cachet of being "real" code, but is large, complex, and full-featured. Not only will it have a steep learning curve, but it will be difficult to modify because all of the simple improvements will have already been made. Ailamaki and Hellerstein [1] describe a course that uses PostgreSQL for this purpose. Their students were able to successfully modify the system, but only two system assignments were possible during the semester (one that modified the buffer manager, and one that added an additional query operator).

A pedagogical system has the luxury of being able to include whatever features seem reasonable, and to omit the rest. It can be written with an eye towards being a readable presentation of the essential concepts, instead of just being efficient. However, it runs the risk of being too simple and therefore irrelevant.

The most widely-available pedagogical system is Minibase [2, Chapter 30]. Minibase attempts to have the structure and functionality of a commercial database system, and yet be simple to understand and extend. By trying to balance both concerns, it winds up not being very good at either. (This assessment is echoed in [1].) It has a high learning curve, but without the advantages of an open-source system. It has no multi-user or transaction capability. The suggested projects are similar to those of [1], and apparently no simpler.

Another pedagogical system mentioned in the literature is MinSQL [3]. This system is designed to have heavyweight architecture but lightweight code. That is, the components of the system and their functionality are essentially the same as in a commercial system, but the actual code implements only a small fraction of what it could. For example, instead of implementing all of SQL, the system implements just enough to allow for non-trivial queries.

The advantage to a system such as MinSQL is that it simplifies the learning curve – students are not overwhelmed by irrelevant features and their consequent programming details. Unfortunately, MinSQL was never made public, and so it is not possible to build a course around it.

I developed the SimpleDB database system, independent of MinSQL, but for exactly the same reasons. SimpleDB's primary goal is to be readable, usable, and easily modifiable. As with MinSQL, it has the basic architecture of a commercial database system, but stripped of all unnecessary functionality and using only the simplest algorithms. The system is written in Java, and takes full advantage of Java libraries. For example, it uses Java RMI to handle the client-server issues, and the Java VM to handle thread scheduling.

I have been using SimpleDB (in various incarnations) for over three years in my Database System Internals course. This paper describes SimpleDB, its architecture, and my experience with using it in the course.

2. THE SIMPLEDB SYSTEM

The SimpleDB code comes in three parts:

- The client-side code that contains the JDBC interfaces and implements the JDBC driver.
- The basic server, which provides complete (albeit bare-bones) functionality but ignores efficiency issues.
- Extensions to the basic server that support efficient query processing.

The following subsections address each part.

2.1 The Client-Side Code

A SimpleDB client is a Java program that communicates with the server via JDBC. For example, the code fragment of Figure 1 prints the salary of each employee in the sales department.

```
String   qry = "select sal from EMP " +
               "where dept = 'sales' ";
Driver     d = new SimpleDriver();
Connection c = d.connect("cs.bc.edu");
Statement  s = c.createStatement();
ResultSet  r = s.executeQuery(qry);
while (r.next())
    System.out.println(r.getInt("sal"));
r.close();
c.commit();
```

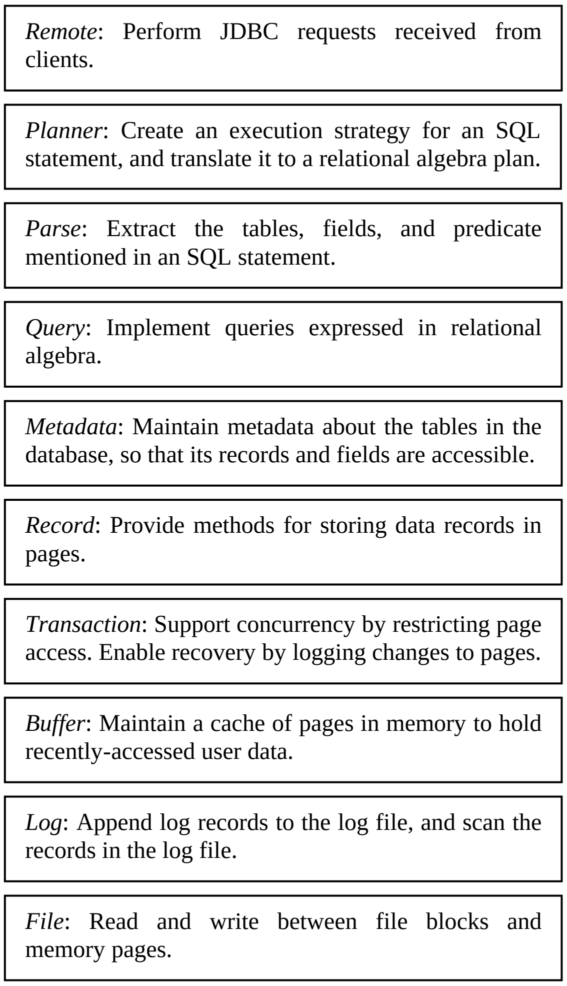
Figure 1: Printing the salary of everyone in the sales dept

The JDBC package *java.sql* defines the interfaces *Driver*, *Connection*, *Statement* and *ResultSet*. The database system is responsible for providing classes that implement these interfaces; in SimpleDB, these classes are named *SimpleDriver*, *SimpleConnection*, etc. The client only needs to know about *SimpleDriver*, but all classes need to be available to it. In most commercial systems, these classes are packaged in a jar file that is added to the client's classpath. SimpleDB does not come with a client-side jar file, but it is an easy (and useful) exercise for the students to create one.

The standard JDBC interfaces have a large number of methods, most of which are peripheral to the understanding of database internals. Therefore, SimpleDB comes with its own version of these interfaces, which contain a small subset of the methods. The advantages are that the SimpleDB code can be smaller and more focused, and that the omitted methods can be implemented as class exercises, if desired.

2.2 The Basic Server

The basic server comprises most of the SimpleDB code. It consists of ten layered components, where each component uses the services of the components below it and provides services to the components above it. These components are displayed in Figure 2. The remainder of this section discusses these components briefly, from the bottom up.



Remote: Perform JDBC requests received from clients.

Planner: Create an execution strategy for an SQL statement, and translate it to a relational algebra plan.

Parse: Extract the tables, fields, and predicate mentioned in an SQL statement.

Query: Implement queries expressed in relational algebra.

Metadata: Maintain metadata about the tables in the database, so that its records and fields are accessible.

Record: Provide methods for storing data records in pages.

Transaction: Support concurrency by restricting page access. Enable recovery by logging changes to pages.

Buffer: Maintain a cache of pages in memory to hold recently-accessed user data.

Log: Append log records to the log file, and scan the records in the log file.

File: Read and write between file blocks and memory pages.

Figure 2: The components of the basic SimpleDB server

The file manager supports access to the various data files used by SimpleDB: a file for each table, the index files, some catalog files, and a log file. The file manager API contains methods for random-access reading and writing of blocks. Higher-level components see the database as a collection of blocks on disk, where a block contains a fixed number of bytes.

The log manager is responsible for maintaining the log file. Its API contains methods to write a log record to the file, and to iterate backwards through the records in the log file.

The buffer manager is responsible for the in-memory storage of pages, where a page holds the contents of a block. Its API contains methods to pin a buffer to a block, to flush a buffer to disk, and to get/set a value at an arbitrary location inside of a block. Higher-level components see the database as a collection of in-memory pages of values.

The transaction manager is a wrapper around the buffer manager. It has essentially the same API as the buffer manager, with some additional methods to commit and rollback transactions. The job of the transaction manager is to intercept calls to the buffer manager in order to handle concurrency control and recovery. It treats blocks as the unit of lock granularity, obtaining an slock (or xlock) on the appropriate block whenever a method to get (or set) a value is called. The transaction manager also supports recovery by using write-ahead logging of values; when a method to set a value is called, the transaction manager writes the old value into the log before telling the buffer manager to write the new value to the page. Higher-level components still see the database as a collection of pages of values, but with methods that ensure safety and serializability.

The record manager is responsible for formatting a block into fixed-length, unspanned records. Its API contains methods to iterate through all of the records in a file. The record manager hides the block structure of the database. Higher-level components see the database as a collection of files, each containing a sequence of records.

The metadata manager stores schema information in catalog files. Its API contains methods to create a new table given a schema, and to retrieve the schema of an existing table. The metadata manager hides the physical characteristics of the database. Higher-level components see the database as a collection of tables and indexes, each containing a sequence of records.

The query processor implements query trees that can be composed from the relational algebra operators *select*, *project*, and *product*. Its API contains methods to create a query tree and to iterate through it.

The parser recognizes a stripped-down subset of SQL, using recursive descent. The language corresponds to select-project-join queries having very simple predicates. There are no Boolean operators except “and”, no comparisons except “=”, no arithmetic or built-in functions, no grouping, no renaming, etc.

The planner builds a query plan from the parsed representation of the query. The plan is the simplest possible: It takes the product of the mentioned tables (in the order mentioned), followed by a select operation using the where-clause predicate, followed by a projection on the output fields.

Finally, the remote interface implements a small subset of the JDBC API. The key method is *Statement.executeQuery*, which calls the parser and planner to construct the query tree and passes it to the *ResultSet* object for traversal. All of the network communication is taken care of by Java RMI.

2.3 Efficiency Extensions

The basic query processor only knows about three relational operators. It doesn't know how to use indexing, nor can it handle sorting or grouping. Moreover, the iterator implementations are as simple as possible – most notably, the implementation of *product* uses nested loops.

These algorithms are, of course, remarkably inefficient. But they also have a simplicity that allows students to focus on the flow of control in the execution of a query tree. Students tend to have difficulty grasping how a query tree of iterators works, and so clarity is more important than efficiency at this point.

The basic planner is equally simple. It does not try to perform joins, or push selections, or optimize join order. The advantage is again clarity over efficiency. A trivial planner allows students to focus exclusively on how the translation from SQL to relational algebra works.

But efficiency, of course, is critical for a database system. Once students understand the basic server, it can be extended with four components to improve efficiency:

- Support for indexing.
- Sorting, and operators that rely on sorting (such as aggregation, duplicate removal, and mergejoin).
- Sophisticated buffer allocation.
- Query optimization.

The indexing component implements both B-tree and static hash indexes, and provides implementations of the *indexselect* and *indexjoin* operators.

The sorting component provides a *sort* operator, implemented using a simple mergesort algorithm. It also uses the *sort* operator to implement *groupby* and *mergejoin* operators.

The buffer allocation component modifies the *sort* and *product* operators to take maximum advantage of available buffers.

The query optimization component implements an intelligent planner. The planner uses a greedy optimization algorithm, and can be configured to use the various efficient operators (e.g. *mergejoin* or *indexjoin* instead of *product*) when possible.

3. TEACHING THE COURSE

3.1 Topics

The Database System Internals course is geared towards junior/senior undergraduates. For practical reasons, I do not require that students take an introductory database course first, but certainly such a prerequisite would help the class move faster. Instead, I spend the first week of the course teaching the fundamentals of table creation in SQL, the relational algebra operators *select*, *project*, *product*, and *join*, and the corresponding queries in SQL. This basic literacy can carry students very far through the course; advanced topics (such as indexing and aggregation) can be dealt with on an as-needed basis.

The course is structured into three parts:

1. How to use a database system.
2. The basic architecture of a database system.
3. Efficient query processing.

Part 1 covers the use of relational databases via basic SQL, and the principles of client-server interaction using JDBC. It also examines details of client-server communication, showing students how a database driver can be built using RMI.

Part 2 considers the internals of the basic database server. For each database component I explain the issues, consider various designs, and describe the design decisions made by SimpleDB. As a result, the students can see exactly what services each component provides, and how it interacts with lower-level components to get what it needs. By the end of this part, students have witnessed the gradual development of a simple but completely functional system.

Part 3 considers efficiency issues. This part studies the sophisticated techniques and algorithms that can replace the simple design choices made in Part 2. The topics in this part parallel the extensions to the basic SimpleDB server, and include indexing, sort-based techniques, advanced use of buffers, and query optimization.

This organization introduces topics in a somewhat different order from a typical database course. For example:

- *Transaction processing is treated relatively early*. Most database courses introduce transactions towards the end, which gives them a sense of being an "add on". I think it is better to discuss a transaction as the low-level concept it is, in order to give the sense that transactions are an integral, tightly-integrated part of a database system.
- *Indexing and sorting are treated relatively late*. I think it is better to wait until part 3, so that these topics can be introduced as solutions to efficiency problems. At that point, students will have a firm understanding of what those problems are, how these new concepts address the problems, and how they all fit into the overall system architecture.

3.2 Assignments

Because SimpleDB implements only a tiny portion of SQL using the simplest algorithms, there are numerous opportunities for students to extend the system with additional features and more efficient algorithms.

Homework assignments are the focal point of the course. I give weekly assignments, to be done individually. Each assignment involves modifying SimpleDB in some way, and may also include some traditional pencil-and-paper exercises.

In the most recent offering of the course, nine assignments were given during the 13-week semester, having the following tasks:

1. *Write a JDBC program*. The program performed some basic database retrieval from the SimpleDB server, and was a good warmup assignment.
2. *Implement authentication*. SimpleDB allows JDBC clients to connect anonymously. Students had to modify the driver's *connect* method to take a username and password, and modify the server to perform the authentication.
3. *Modify the buffer manager*. Students not only implemented a different page replacement algorithm, they also modified how the buffer manager organized the buffer pool.
4. *Modify deadlock detection*. SimpleDB uses a timeout mechanism to detect deadlocks. Students replaced it with the wait-die algorithm.
5. *Add non-quiescent checkpointing*. Students had to define a new type of checkpoint log record, modify the server to

periodically add the checkpoint record to the log, and modify the recovery code to use it.

6. *Add the ability to scan records backwards*. SimpleDB only implements the recordset methods *beforeFirst* and *next*. Students added the methods *afterLast* and *previous* to the JDBC *RecordSet* interface, and modified the record manager to support these methods.

7. *Implement new relational algebra operators*. Students wrote code for the *union* and *rename* operators. This code included the iteration methods *beforeFirst*, *next*, *afterLast*, and *previous*, as well as methods to estimate block accesses and output records.

8. *Modify SQL*. The SimpleDB version of SQL does not support range variables. Students had to modify the SQL parser (and grammar) to recognize the AS keyword in the *from*-clause, and make corresponding modifications to the planner.

9. *Implement prepared statements*. (This was a 2-week assignment.) Students needed to implement the JDBC *PreparedStatement* interface. This involved modifying numerous portions of the server, and was a challenging and interesting final project.

The scope of these assignments is remarkable, and is possible because of the bare-bones nature of SimpleDB. The code is minimal and easy to read, which makes for an easy learning curve. Each component of SimpleDB makes heavy use of the methods defined in its next-lower component, and so a strong knowledge of the API can lead to very few lines of code. In fact, many of the assignments could be completed with less than 100 lines of code.

3.3 Pedagogical Observations

A database system is a remarkable piece of software. Its internals cover diverse topics, such as file systems, multi-user transaction processing, data structures, language interpreters, and optimization algorithms. Many of the seniors have encountered these topics separately in other courses, and are pleased to be able to study a system in which it all fits together. In this sense, the course acts as a capstone course for those students.

In order to solve the assignments, the student usually has to first understand where the affected code is, plan the modification, and only then write the necessary (and short) code. This approach to coding comes as a shock to some students, who are used to being able to grind out reams of code with very little thought or planning.

The course is exceptionally code-intensive, and the students spend a lot of time both reading and writing code. The SimpleDB code attempts to be elegant, and many students commented on how they learned the value of good programming style from trying to imitate it. Some of the better students have been motivated to “out-elegant” the instructor in their solutions to the assignments.

This course is one of the only courses in our curriculum in which students have the opportunity to work with a large, functional system. Throughout their coursework, students typically encounter toy systems or simulators, and never get to see how everything fits together. Particularly interesting is their approach to Java packages. The early resistance towards packages eventually gives way to acceptance (since they have no choice but to use them), and then to appreciation as they realize that focusing on a single package is a lot easier than having to wade through all possible SimpleDB classes.

4. CONCLUSION

The SimpleDB database server was written to help students in a database systems internals course. It has two purposes:

- to give students an easily-understood example of a real database system;
- to give students a vehicle for doing meaningful hands-on programming assignments.

My experience has been that the system has fulfilled these purposes very well. The system has also turned out to be valuable in ways unrelated to the study of database systems. It gives students experience with grappling with a large system, both in trying to understand it and to modify it. And it provides a practical, “capstone” example of numerous theoretical concepts that students have encountered in other courses.

The basic SimpleDB server consists of about 3,500 lines of Java code (not including the JavaDoc comments), and the efficiency extensions are about half that size. The URL *www.cs.bc.edu/~sciore/simpledb/intro.html* contains instructions for downloading and configuring the system, as well as a pointer to the website for my Database Systems Internals course.

5. REFERENCES

[1] Ailamaki, A., and Hellerstein, J. Exposing Undergraduate Students to Database System Internals. *ACM SIGMOD Record, 32, 3* (September 2003), 18-20.

[2] Ramakrishnan, R. and Gehrke, J. *Database Management Systems (Third Edition)*. McGraw-Hill, Boston, 2003.

[3] Swart, G. MinSQL: A Simple Componentized Database for the Classroom. In *Proceedings of the 2nd international conference on Principles and Practice of Programming in Java* (Kilkenny City, Ireland, June 16-18, 2003). ACM International Conference Proceeding Series Vol. 42, 2003, 129-132.