Comparing the Performance of Open Source and Proprietary Relational Database Management Systems

Dissertation

Submitted to Northcentral University

Graduate Faculty of the School of Business and Technology Management
In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

By

SEAN STEVEN COATES

Prescott Valley, Arizona
August 2009

UMI Number: 3386054

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI®
Dissertation Publishing

UMI 3386054
Copyright 2010 by ProQuest LLC.
All rights reserved. This edition of the work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest®

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Copyright 2009

Sean Steven Coates

APPROVAL PAGE

Comparing the Performance of Open Source and Proprietary Relational Database Management Systems

by

Sean Steven Coates

Approved by:

James Neiman Ph. D. 10/31/09

Chair: James Neiman, Ph.D. Date

Member: Michael Ewald, Ph.D.

Member: Efosa Osayamwen, Ph.D.

Certified by:

Lynn Payne 11/16/09

School Dean: Lynn Payne, Ph.D. Date

ABSTRACT

The relative performance and scalability of open source and proprietary relational database systems (RDBMS) were examined using a newly constructed suite of benchmark case tests. Technology managers can save money on software licenses if they switch to open source products, but many have not done so because of concerns about the performance of open source products relative to commercial products. The relative performance and scalability of some of the most popular open source and proprietary RDBMS products was quantitatively compared. A benchmark case was constructed to measure three aspects of RDBMS performance: batch load, transaction processing, and report generation. The benchmark scores for proprietary database products were higher than for open source database products. The differences in performance and scalability were not enough to justify the much higher cost of proprietary database products except in cases where the cost of a proprietary solution would not be a major to an individual technology manager. Future researchers run the benchmark on a different platform and examine the performance of newer versions of the RDBMS products reviewed here or different database products.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

The use of open source technology is having a major impact on businesses today, as it provides technology managers with the ability to use software applications similar to proprietary products at a fraction of the cost (Boulton, 2003; Cohen, 2003; Dickerson, 2003; Hicks, 2002; Krill, 2002; LaMonica, 2005; Silwa, 2005). One of the more expensive proprietary software products technology managers are compelled to buy is a relational database management system (RDBMS). There are many open source RDBMS products available, but many technology managers feel that the proprietary products are superior in most ways, thus justifying the large cost (Boulton, 2003). The performance and scalability of proprietary and open source RDBMS products available today was compared in this research. A process to compare the performance of RDBMS products was designed and developed using the same operating system and server hardware. To compare the RDBMS products, benchmark tests were created and used. An analysis of the benchmark testing results was performed to determine, cost factors aside, whether open source database systems can perform as well as proprietary database systems when executing common tasks.

Background

Technology managers may be able to reduce capital and operating expenditures by using open source RDBMS products, but they do not know how well the open source RDBMS products available today compare to the proprietary RDBMS products. Proprietary RDBMS products cost between $5,000 and $40,000 per processor for the software license, while open source products do not require a software license (Boulton, 2003). The more servers and processors a company has, the greater the potential savings by using open source RDBMS products. For example, in an environment with 100 servers with two processors each running database software that costs $25,000 per processor, the cost of the software would be $5,000,000 and annual maintenance fees, typically 20%, would run $1,000,000 per year, a significant cost for most firms. The switch to open source products can only be justified if the performance and scalability of the open source RDBMS products is close to or better than the proprietary RDBMS products.

Problem Statement

The problem to be addressed is how to compare the performance and scalability of open source and proprietary RDBMS products. Technology managers would like to be able to cut costs RDBMS licensing by switching to open source, but this can only be justified if the performance of open source products is comparable. Over 67% of technology managers seek out open source solutions to save on costs (D'Agostino, 2005). Technology managers facing this decision need a method of comparing open source and proprietary database products. Historically, researchers who put together benchmarks to measure database performance focused on either open source or proprietary RDBMS products, depending on their target audience. To compare both, a new analysis was needed. This required the construction of a new benchmark, a suite of tests flexible enough that it could be run a wide range of open source and proprietary database systems.

Purpose

The primary purpose of this quantitative study was to determine if technology managers should consider open source database systems acceptable substitutes for proprietary database systems based on the criteria of relative performance and scalability, which were measured by running a benchmark case against several proprietary and open source database systems and comparing the results. If an analysis of the benchmark tests shows that open source databases can significantly outperform or perform as well as proprietary databases, then information technology managers should consider open source databases for all applications where performance is a primary concern. If the analysis shows that open source databases do not outperform the proprietary databases, then technology managers may want to avoid open source databases in situations where performance is a primary concern. Finally, the performance of individual open source databases was compared to other open source databases, to show which underperforming database systems have areas for improvement.

Theoretical Framework

There are few published articles that measure the performance of open source RDBMS products. One of the most widely published results was run by Dyck (2002), in which he compared several database products. One of the problems with Dyck's testing is that he did not hold his external variables constant; there were different operating systems involved, and different methods of accessing the database.

The best example of constructing a benchmark was done by DeWitt (1993) with his Wisconsin benchmark. DeWitt's work eventually led to the development of the TPC-A benchmark (Transaction Performance Processing Council, 1992). The TPC-A and other benchmarks created by the Transaction Performance Processing Council do not reflect real-life database activity, as they are not modeled after a real-life application.

Strandell (2003) developed a benchmark based on activities in the telecommunications industry, which resulted in a highly specialized benchmark that was not useful for applications outside that industry. Still, he was very helpful in showing the kind of work an individual could put together, as most benchmarks today are developed by large organizations like TPC.

This research addressed these concerns through its unique design. All of the testing was performed on a single platform and operating system, reducing the number of external variables. The benchmark suite simulated an electronic commerce application, providing a more realistic load to the databases tested. The benchmark suite used very general terms, which could easily be applied to a wide range of industries. These choices were made to improve upon previous efforts at database benchmarks without making the system overly complex.

Research Questions

Technology managers would like to cut costs by using open source RDBMS products, but they do not know how well the open source RDBMS products available today compare to the proprietary RDBMS products (Florescu & Kossman, 2009). Proprietary RDBMS products cost between $5,000 and $40,000 per processor for the software license, while open source products do not require a software license (Boulton, 2003). Open source products are only a viable alternative if their performance and scalability are comparable to the proprietary products. In order to enable this comparison, a benchmark case was created and run against several popular open source and proprietary RDBMS products. The results were analyzed to find a measure of the relative performance and scalability of open source and proprietary RDBMS products, answering the following research questions:

1. On a server with one processor core, to what extent, if any, does the performance of open source RDBMS products, on average, equal or exceed the performance of proprietary RDBMS products, on average, when run on the same operating system and hardware?
2. On a server with four processor cores, to what extent, if any, does the performance of open source RDBMS products, on average, equal or exceed the performance of proprietary RDBMS products, on average, when run on the same operating system and hardware?
3. To what extent, if any, does the scalability, from one processor to four processor cores, of open source RDBMS products, on average, equal or exceed the scalability of proprietary RDBMS products, on average, when run on the same operating system and hardware?

Nature of the Study

The relative performance and scalability of open source and proprietary RDBMS products was compared. This comparison required the acquisition of the database products being examined, the testing platform, a data model, test data, and a suite of benchmark case tests. For the proprietary products, evaluation licenses were acquired. The open source products were freely available. The testing platform was a small server running Linux®, and the same operating system and server configuration was used for all of the products tested. The data model, test data, and benchmark case tests were constructed as a part this research.

The dependent variables were the individual database products being reviewed (X_d), the number of active processor cores in the system during a test (X_p), the type of test (X_t) and the number of the individual test run (X_r). For each combination of these dependent variables, a duration was measured ($Y_{d,p,t,r}$). These durations were then averaged and combined to provide benchmark scores. The scores were normalized for simpler comparison.

All of the tests were performed using the same physical server equipment, to provide a constant and fair environment for comparison. The server host was restarted before each test to ensure that no other processes would be competing for server resources, and that no memory of prior tests would be cached in the system RAM. Each benchmark case started with the same data, read in from the same flat files used for all of the database products.

Significance of the Study

There have been few published reports comparing the performance of RDBMS products, so having a new comparison is beneficial to those technology managers who have to select one of these products for a new application. This study also includes the details of the construction of the benchmark suite so that technology managers could run the tests on their own systems. The methodology presented here can also be adapted to newer versions of the RDBMS products, which provides technology managers with the ability to create more current numbers.

For those technology managers facing budgetary challenges, the results presented in this research will provide an estimation of the kind of performance they can expect if they switch from expensive proprietary solutions to the much cheaper open source solutions. Even if open source database products are somewhat slower than proprietary products, the performance may be acceptable to technology managers for some applications.

Assuming that the open source database products performed well in the benchmark results, information technology managers should consider open source database products more seriously when it comes time to select a database for a new application. Today, many managers do not consider open source databases because they simply assume they are inferior, and that part of the cost of the proprietary products is for higher performance (Biggs, 2002). If RDBMS products being proprietary are not a reliable indicator of higher performance, then many managers could potentially save hundreds of thousands of dollars, or more, on licensing fees by switching to open source databases. Technology managers that are trying to implement large computing grids and facing high per-processor or per-core licensing fees could save even more money. Open source database products are not completely free, as there are the costs of third-party support contracts, training, and possibly new personnel. The cost savings is in the license fees, which can be substantial in large environments.

Technology managers at many companies have adopted Linux, an open source operating system, and are becoming comfortable with other open source technology (Silwa, 2005). Some managers are even beginning to deploy open source database systems like MySQL and PostgreSQL in order to save on licensing costs (Mears, 2005). Technology managers would also benefit from examining the results of a benchmark case run on a variety of open source and proprietary database systems. Technology managers in general should be interested because the licensing costs of proprietary database systems are rather significant; in many instances, the investment in database technology is so great that costs of switching to a different proprietary platform are insurmountable (Dyck, 2002).

An examination of the benchmark case results also measures the scalability of the database systems, which is the rate of improvement in performance as processor cores are added to the system. Scalability is important because open-source databases do not have per-processor licenses fees, and for some managers it may be, for example, more cost efficient to run an open source database system with four processors than a proprietary system with two processors. With the newly available Intel® (Alfs, 2007) and AMD™ (AMD, 2006) quad-core processors, ordinary systems will have 16 cores or more (Gillespie, 2007). It becomes apparent that the cost of licensing for proprietary databases quickly surpasses that of the hardware. If the performance of open source database systems can be shown to scale up with the number of processors on the system, then it would provide another good reason for technology managers to consider them as an acceptable replacement for expensive proprietary solutions.

While technology managers are always concerned with software expenses, using open source alternatives to traditional proprietary RDBMS products can only be acceptable if the performance and scalability of the open source products is comparable to the competing proprietary products.

Definitions

Included are definitions of some key terms used in this research. While some of the definitions may be common, specific meanings are included here to avoid any uncertainty. Most of the terms are related to database technologies.

RDBMS. A relational database management system (RDBMS) is a system for storing data where data pertaining to an entity is stored in a table, and data for each instance of such entity is stored as a row in the table. Furthermore, attributes describing the entity are stored in columns that can be addressed independently, enabling data in tables to be joined together based on relations which are based on similar columns between tables (Elmasri & Navathe, 1994). RDBMS products are defined as belonging to one of two groups: proprietary and open source. Both types of products have commercial businesses built around them based on the support of their products. Using the relations between the tables, database queries can be formed by joining tables together based on the relations and the constraints of the query. Database users perform queries by using SQL, or structured query language. All RDBMS products use SQL to query their data (Silberschatz, Korth, & Sudarshan, 2003).

Processor core. A processor core is a processing unit within a computer, recognized by the operating system as a separate processor. Processors are generally capable of performing calculations independently of each other (Silbertschatz, Korth, and Sudarshan, 2003).

Open source. The authors of open source products make their source code freely available, which means that there is no cost to acquire the software (Chen, 2002).

Proprietary. The developers of proprietary products generally do not make their source available, and they charge a license fee for their software (Donston, 2002).

Speedup. An important factor to consider when comparing databases systems is the difference between speedup and scaleup. Basically, speedup is a measure of how a system can complete the same tasks in less time by adding processor cores to it. Linear speedup is the desirable goal here, with each additional processor core reducing the time spent by a proportional amount (DeWitt, 1993).

Scaleup. Scaleup is a measure of how much more data can be processed in the same amount of time when processors are added to the server (DeWitt, 1993).

Benchmark case. A benchmark case is defined as a set of database activities that are performed. A benchmark case test is a particular run of the activities in a benchmark case. The duration of a benchmark case test is the time it takes to perform all of the necessary activities in the case (Gray, 1993).

Benchmark case test. A benchmark case test is an individual test within the set of tests that make up the benchmark case (Gray, 1993).

Benchmark case test run. An individual run of a test. A test is typically run multiple times (Gray, 1993).

Performance. Performance is defined as the average duration of the benchmark case tests for a particular server configuration (Gray, 1993).

Scalability. Scalability is the ratio between the performance of a system in one configuration and the performance of a system in a lesser configuration (Gray, 1993).

Primary key. A database table is made up of rows, with each row representing an instance of an entity; for example, in a table of user accounts, each row would represent one user account. Individual rows are identified with a unique identifier that is made up of one or more columns of data known as the primary key.

Foreign key. When one data in one table refers to data in another table, it has a relation through a foreign key. The foreign key refers to the primary key in the other table. For example, in a table of transactions, one of the columns may refer to the user account associated with the transaction; instead of containing all of the user account information, one can put in a foreign key that refers to an entry the user accounts table. Foreign keys are not unique; for example, multiple transactions can refer to the same user account.

Summary

A benchmark case was constructed that measured the performance of a database product in three different areas: database load, transaction processing, and generation of reports for decision support. The benchmark case was run for several RDBMS products, all of which will use the same hardware and operating system in order that the hardware and operating system would not be factors in the results of the comparison.

The costs of the various database products were not addressed because the costs vary depending on the user. For example, technology managers in most companies would find installing several copies of Oracle® to be very expensive, as the licensing and support costs are rather high compared to other products (Pallatto, 2005). In addition, the same managers may find it difficult and expensive to locate and hire experienced Oracle® database administrators, due to the shortage of trained personnel (Chabrow, 2008). Technology managers in other companies may already have an Oracle® site license or extra licenses available, greatly reducing the incremental cost of installing another Oracle® database. The actual cost of database product licenses, maintenance, and the personnel to support them will vary from firm to firm. The same is true for open source databases; some firms may have people already on the staff capable of supporting the product, while managers at other firms would need to purchase maintenance agreements from third party organizations to support the database. Therefore, the cost of the database is not be a factor in the measurement of the performance of RDBMS products in this research.

When a database vendor does not have a well-performing database, their marketing managers may create price-to-performance measures to use in their advertising (Register Research, 2003), but for the reasons just discussed, their measures can be misleading at best. Another tactic used by marketing managers at a database vendor is to only report portions of a benchmark result that look favorable on their products (Caniano, 1988; Scannell, 2000). Thus, one generally cannot trust benchmark reports from the vendors themselves, and it is best if the tests are run by an independent party.

A new way to measure the relative performance and scalability of RDBMS products was provided through the construction of a benchmark case. How open source and proprietary RDBMS products currently compare was shown by running the tests in the benchmark case against some of the more popular database products available in the market today. Both open source and proprietary RDBMS products continue to evolve and improve, however, and will perform differently in the future. In addition, the performance of individual database products will be different on other hardware platforms and different operating systems. Future researchers may do a similar analysis using a different environment. In particular, computers in the future will have eight or more processor cores, exponentially more memory (Coffee, 2005), and access to solid state storage technology (Mitchell, 2006).

To assist technology managers who are considering open source RDBMS products, a benchmark case was run and the results analyzed to measure to what extent, if any, the proprietary RDBMS products exceed the open source RDBMS products in the areas of relative performance and scalability. The details about the construction and use of the benchmark will provide technology managers with the ability to adapt the benchmark suite to their own systems and applications. If the performance of open source database products is acceptable to technology managers, they could realize significant cost savings by selecting open source RDBMS products over proprietary RDBMS products.

CHAPTER 2: LITERATURE REVIEW

The literature review is divided into three major sections: open source software, relational database management systems, and database benchmarks. There is little published literature regarding database benchmarks, but what is available is listed here.

Open Source Software

In recent years, open source software has become more readily accepted for use in the enterprise by technology managers of all sizes of companies, as was shown in a recent survey (D'Agostino, 2005). For the survey, 235 senior information technology executives were questioned about the use of open source software at their companies. The primary reason for using open source software was for the lowered costs, according to 67% of the respondents. It was also reported that 72% of executives planned to expand the use of open source software at their companies. The most widely used open source package was the Linux operating system (by 87% of respondents), followed by the webserver Apache (65%), the MySQL RDBMS (58%), and the Firefox web browser (48%). Cost savings was not the only motivation for using open source applications; 64% of executives surveyed felt that the use of open source software in their enterprise had the potential to give them a distinct competitive advantage over their rival firms. Other reasons for using open source software included avoiding vendor lock-in, improved security, better functionality, and the ease of customization. Furthermore, 57% of technology executives found the quality, capability and ease of use of open source systems to be so good that they would be willing to pay for them if they were proprietary products.

There have been hundreds of open source software projects, many of which have been successful at gaining popularity and becoming widely used. Typically, an open source software project gets started by an individual need, or as Raymond (2001) put it, "Every good work of software starts by scratching a developer's itch." Raymond told the story of the open source operating system Linux, from its beginnings as Linus Torvald's pet project to the modern operating system in use at most companies today. According to Raymond, when there is a strong enough perceived need for a product, a community of developers can gather and work in concert to develop open source software that rivals the finest commercial products. By leveraging the technologies that ease communication and collaboration over the Internet, open source projects frequently find themselves drawing upon a global pool of developers with similar interests.

Open source software typically has no license fees, making it an attractive option to technology managers when making purchasing decisions under budget constraints. Over the past few years, open source products have been appearing more frequently in large enterprise organizations for several reasons. The most common reason is the need to cut licensing costs, making it easier to justify the consideration of open source alternatives to expensive proprietary software (Chen, 2002). Furthermore, the wide support for Linux in the software industry has proven that the open software model can be viable. Every year there is an increasing amount of applications available on the Linux operating system (Chen, 2002). Additionally, many managers have noted the success of the Apache webserver software, an open source product that is used in a majority of organizations. MySQL, an open source database product, has also proven to be quite successful in gaining market share. Many large vendors have committed to support open source software products by integrating them into their platforms and providing robust levels of support (Chen, 2002).

The quality of open source software has always been a concern to those who consider using it. Aberdour (2007) discussed the benefits of the open source software model with regards to the quality of the code. Open source software projects typically have a small core set of developers that maintain the code, assisted by layers of other contributors. Each layer of contributors surrounds the core much like an onion. The closer to the core the contributing developers are, the more effects they will have on the software. The outer layer consists of the users of the software who report problems to the project, people who often never examine the actual source. Because of the many layers of developers writing, improving, examining, and using the software, problems are identified and fixed in a timely manner. Aberdour also notes that not all open source software projects are equal, and some have much higher quality than others.

Open source software development often leads to innovation in features and techniques (Ebert, 2007). Because the open source model allows new ideas to be easily shared with the development community surrounding a product, new ideas quickly get developed into new features. Sometimes the features are so popular that the proprietary products take notice and, after some delay, incorporate the features into their own products. Ebert discussed Asterisk, a telephony software package, as a good example where the open source model has led to a lot of innovation. Prior to Asterisk, the telephony market was dominated by a small number of large firms. Now, many new suites of products built around Asterisk are being offered to small businesses, with many new features compared to their legacy phone systems.

With the source code being openly available to anyone, some people may have concerns about the security of the software. Malevolent hackers could use the open access to the software to more easily find a means of penetration. Also, there is some concern about hackers introducing security holes as part of the community development process. However, some researchers (Witten, Landwehr, & Caloyannides, 2001) believe that by having a much larger universe of people maintaining and testing the code, security should generally be improved. In addition, the software should be easier to repair in the event that a weakness is found, precisely because such a wide variety of people are familiar with the source code.

Relational Database Management Systems

The concept of an RDBMS can be simply described a tool that does three things: it allows one to add data to it, it stores the data, and it allows you retrieve and work with the data. Loney & Koch (2000) further described an RDBMS as storing information in tables, with each table having one or more columns. Data can be stored and retrieved from the tables by using Structured Query Language (SQL). Because SQL can be used to extract data from multiple tables at once by joining the data using similar, related columns, the database is said to be relational. That is, two tables are related if a join operation can be performed using similar columns. Relations between tables allow a database designer to minimize redundant data through a process of normalization. Typically, an RDBMS consists of a complete suite of tools for the management of data, making them far more useful than simply storing data in flat files (Loney & Koch, 2000).

The idea for using a relational model to represent data was originally presented by Codd (1970). Prior to the research based on Codd's paper, databases had a hierarchical layout, with each data element being the child of a parent element (Oppel, 2004). Codd presented not only the idea of laying data for entities out in related tables, but he also developed the concepts of normalization and a mathematical language for querying data. The mathematical nature of Codd's querying language restricted the popularity of relational databases in the early years.

Starting in 1974, IBM began a major research project to build the first RDBMS, based on Codd's work. One of the products of IBM's research was the database itself, called System/R, and a querying language called SEQUEL, which was an acronym for Structured English Query Language. Due to legal problems, SEQUEL was later changed to SQL (for Structured Query Language). In 1978, IBM was able to distribute their new RDBMS products to some customer sites, where end users began to develop experience with RDBMS concepts (Groff, 2002).

In 1979, Relational Software, Inc. released the first commercial RDBMS product, Oracle. The engineers at Relational Software had been following the research at IBM closely and were able to develop a commercial product a full two years ahead of IBM. The original version of Oracle had a limited feature set, much like early versions of open source RDBMS products (Groff, 2002).

Another research effort underway at the University of California's Berkeley computer labs led to the creation of an RDBMS called Ingres. The Ingres team created a query language called QUEL that was more structured and had fewer similarities to normal English when compared to SQL. In 1980, Michael Stonebreaker, Eugene Wong, and others left Berkeley and founded Relational Technology, Inc. to develop a commercial version of Ingres that would compete with Oracle. Although Ingres was technically superior to Oracle, the SQL language was easier to use and Oracle had better marketing. As a result, Ingres adopted SQL in 1986, but it was too late to compete effectively with Oracle. Faced with dwindling market share, the Ingres technology was eventually sold to Computer Associates 1994 (Groff, 2002).

The results of the System/R research project were developed by IBM into a commercial RDBMS product called SQL/Data System (SQL/DS). The product was released to customers in 1982 for use on IBM mainframe computer systems running VM/CMS. In 1985 IBM released Database 2 (DB2), for use on IBM mainframe computer systems running the MVS operating system, which had become more widely used than VM/CMS. DB2 used SQL, and with both Oracle and DB2 using it, SQL became the industry standard database language. IBM eventually made versions of DB2 for all systems in its product line, as well as all major versions of UNIX, including those of its competitors, Hewlett-Packard and Sun Microsystems (Groff, 2002).

In 1984 Bob Epstein, who had worked on the Ingres project in its early years, founded Sybase, a company that created an RDBMS product for use on minicomputers from Hewlett-Packard, Sun Microsystems, and other UNIX platforms. Sybase later developed a partnership with Microsoft to create a version for OS/2. Although Microsoft and Sybase later dissolved their partnership, each was able to take the current code and develop it commercially, leading to Sybase's Adaptive Server Enterprise and Microsoft's SQL Server (Oppel, 2004).

One of the earliest examples of an open source RDBMS was MiniSQL, written in 1994 by David Hughes as part of his dissertation at Bond University in Australia (Yarger, Reese, & King, 1999). MiniSQL was the first open source RDBMS to use SQL to query data. Prior to the development of MiniSQL, Postgres, an open source derivative of the commercial RDBMS Ingres, was available, but at the time it used a unique query language called PostQUEL. Originally, the purpose of MiniSQL was to translate SQL commands into PostQUEL for storage in a Postgres database. Over time the translator proved to be inefficient, and he replaced PostQUEL and Postgres with his own database engine. The result of his efforts was mSQL.

MySQL was created partly as a reaction to some of the weaknesses of the early versions of mSQL. According to Yarger, Reese, & King (1999), Michael Widenius had created a database storage engine and was looking for a SQL front end to use with it so he could develop database-driven web sites. After discussing the matter with David Hughes, he decided that mSQL was lacking in features and functionality, which led him to create his own front end and by 1995 MySQL 1.0 was completed. Because MySQL had more features than mSQL, it quickly grew to be significantly more popular than mSQL (Yarger, Reese, & King, 1999).

Also in 1995, two of Michael Stonebreaker's graduate students, Andrew Yu and Jolly Chen, designed a SQL interface to Postgres, eliminating the need to use PostQUEL. They called the new open source RDBMS PostgreSQL, and it had more features than early versions of MySQL, including transactions, triggers, and subselects. The enhanced feature set led to greatly increased popularity for PostgreSQL, especially for those applications that required transactional support. MySQL was generally found to be easier to install and use, so it retained its own popularity for applications that did not require a lot of transactional support (Yarger, Reese, & King, 1999).

In 1984, Jim Starkey created Interbase, an RDBMS product for use on the PC platform. Interbase was later sold to Ashton-Tate, and when Ashton-Tate had financial difficulties, the company was then acquired by Borland. For a brief period in 2000, Borland made a version of InterBase available as open source, which led to a derivative version of the code tree which became known as the Firebird RDBMS (Niccolai, 2006). Although Firebird is not as well-known as PostgreSQL or MySQL, it continues to win awards for being a great open source RDBMS (Martens, 2007). Developers like Firebird for the compact size of its database and straightforward installation on PCs running Windows (Cox, 2004).

Another open source RDBMS that is gaining in popularity is EnterpriseDB, a product developed on a PostgreSQL base and enhanced to make it highly competitive with Oracle. EnterpriseDB is now compatible with the Oracle Call Interface, allowing programmers to switch from Oracle to EnterpriseDB with little or no changes. It also works with commercial enterprise software from vendors such as SAP and PeopleSoft (Niccolai, 2007). Niccolai also quoted an analyst at Forrester Research, Inc., Noel Yuhanna:

> EnterpriseDB is built on top of PostgreSQL which is a proven enterprise DBMS for decades, therefore has reliability and robustness, besides offers good overall performance and scalability.
>
> We find that all customers that are looking to save money on database management, should look at EnterpriseDB, along with other open source databases such as MySQL and Ingres (Niccolai, 2007, para. 10, 11).

Although EnterpriseDB is still relatively small and only recently acquired its 100th customer, Oracle's acquisitions of several application vendors could lead many technology managers to seek alternatives, and thus avoid being locked into an all-Oracle environment. Then again, some customers prefer having to deal with only one vendor (Niccolai, 2007).

MaxDB is the result of a joint venture started in 2004 between SAP AG and MySQL AB. Previously, SAP had developed SAP DB for internal use in its enterprise software products, known as the suite SAP R/3. They market it as a free alternative to using expensive commercial RDBMS software in R/3. By partnering with MySQL AB, they are making the RDBMS available as open source and hope to find a larger audience for it. As part of the deal, they renamed the product from SAP DB to MaxDB (Songini, 2004).

Starting in 1998, most database vendors made their RDBMS products available on the Linux platform (Cornetto, 1998). It became apparent that Linux was gaining in popularity and could eventually be the operating system of choice at most companies. For the first time, all major open source and proprietary RDBMS products were available on the same operating system, making it possible to do more direct comparisons.

Database Benchmarks

There has been little published research comparing the performance of open source and proprietary relational database systems. Previously, other researchers have created and run benchmark cases to analyze various aspects of database systems. What follows is a review of the published research on database benchmarks.

There are many difficulties in making a fair comparison between RDBMS running on different operating systems (Apicella & Biggs, 2000). The main difficulty is that one cannot determine how much the operating system contributes to the performance of the database. In the eWeek testing (Dyck, 2002), they ran their benchmark on Windows® for the Microsoft® SQL Server and on Linux® for the other database vendors. Because they introduced different operating systems as an independent variable, there was no way to tell if the difference in performance was more a result of the different database product or the different operating system, as their research did not use a multi-factor experimental design. After publishing their results, further analysis by employees at Microsoft revealed other differences, due to the web application server technology (Dyck, 2003). The researchers at Microsoft told Dyck that the configuration of the ASP.NET software was not set up to make a fair comparison to the Java Server Pages code the other systems used.

Early benchmark writers included simulations using terminals connected via X.25 networks (Serlin, 1993). The simulations used in the early benchmarks reflected a common configuration found in the retail branch networks of large banks at the time. Unfortunately, the delays intrinsic to a network of remote terminals soon became a limiting factor in the TPC-A benchmark, making it obsolete (Levine, Gray, Kiss, & Kohler, 1993).

Gray and Nyberg (1994) showed, over ten years ago, that servers made out of commodity parts are becoming more and more capable of running OLTP and batch processing workloads. Today, even a common desktop PC is powerful enough to run complex OLTP and batch processing jobs.

The author of the Wisconsin benchmark (DeWitt, 1993) paid particular attention to the issues of speedup and scaleup, even though it was one of the earliest benchmarks. DeWitt and his associates were concerned with improving the performance of a database by adjusting its hardware configuration.

Designing a benchmark is a challenging endeavor typically done by developers in large organizations such as the Transaction Processing Council and their (1992) TPC-A benchmark. The TPC was formed because previous individual attempts encountered many difficulties, and early database researchers sought to combine their experiences when developing the TPC benchmark. One example of an individual's effort to create a benchmark was Strandell (2003), where the author created a benchmark case and ran it on a couple of database systems. Strandell centered the benchmark case on the telecommunications industry. He took the average durations of results that made it into the 90th percentile, thus throwing out any outlier results. It would have been more helpful if he had discussed the witnessed variance of the results, and compared the results of one database system's score to another in a statistically significant way.

In Ailamaki, DeWitt, Hill, & Wood (1999), researchers performed their experiments by running the same benchmark for four databases on the same hardware. To increase confidence in the results, the experiments were repeated several times until the final numbers had a standard deviation of less than 5%, after which the experimenters did not feel the need to provide any other statistical comparisons. From there, they plotted values on a graph, and relied on the small standard deviation and major differences between variables to indicate differences. Ailamaki et al. also found that, after testing four commercial database products for the Windows® platform, database developers could greatly benefit by using the L2 cache more intelligently. As the L2 cache became larger and more commonly available in modern processors, the measured performance of databases using the L2 cache far exceeded those that did not use it. Clearly, benchmark tests are useful in revealing which database systems are not taking advantage of new advances in processor technology.

Several journal articles contained very little or no statistical analysis. One study discussed the relative performance of systems doing batch processing, as well as the relative costs, and how it has changed over the years (Gray & Nyberg, 1994). In their paper they merely reported their numbers in tables, with no mentions of means, standard deviation, variance, or statistical significance.

Statistical analysis of experimental data is not always included in published research. For example, there was a comparison of the various database systems used in libraries, as well as the database vendors' relative market share in 1999 and 2000 (Matoria & Upadhayay, 2002). The data in their report was presented in tables and pie charts, with no statistical analysis.

Poess & Floyd (2000) similarly ignored statistical analysis when discussing new benchmarks TPC-H, TPC-R, and TPC-W. They also reported some of the early results from the TPC benchmarks, but provided no discussion as to how the results compared to each other. There were no confidence intervals on their reported statistics, making it impossible to tell if the differences between scores were statistically significant.

To avoid some of the problems earlier researchers encountered, overly complex benchmark tests should be avoided. Although the database queries in a benchmark case may include many of the common advanced features SQL has to offer, the suite should be simple enough to implement on a wide variety of database products. In the end, moderation is very important in accomplishing successful tests.

Summary

Open source products are being adopted for a wide variety of applications by many technology managers, especially as these products have become more reliable and functional. Many open source database products share the same origins as their proprietary counterparts. There have been few published reports of database benchmarks, and the quantitative detail of these reports has been varied in quality. Technology managers could benefit from a more current analysis, particularly with all of the database vendors releasing new versions of their software in recent years.

CHAPTER 3: RESEARCH METHOD

An examination of the relative performance and scalability of proprietary and open source RDBMS products was needed to provide technology managers with better insight when making purchasing decisions. A suite of benchmark tests were designed and developed to enable measurement of the performance of a given RDBMS. The benchmark case tests were run on each proprietary and open source RDBMS product on a server with one active processor core. The benchmark tests were then repeated with four active processor cores, in order to be able to measure the scalability of each database system.

Technology managers would like to cut costs by using open source RDBMS products, but they do not know how well the open source RDBMS products available today compare to the proprietary RDBMS products. Proprietary RDBMS products cost between $5,000 and $40,000 per processor for the software license, while open source products do not require a software license (Boulton, 2003). A benchmark case was created and run against several popular open source and proprietary RDBMS products. The results were analyzed to find a measure of the relative performance and scalability of open source and proprietary RDBMS products, answering the following research questions:

1. On a server with one processor core, to what extent, if any, does the performance of open source RDBMS products, on average, equal or exceed the performance of proprietary RDBMS products, on average, when run on the same operating system and hardware?

2. On a server with four processor cores, to what extent, if any, does the performance of open source RDBMS products, on average, equal or exceed the performance of proprietary RDBMS products, on average, when run on the same operating system and hardware?
3. To what extent, if any, does the scalability, from one processor to four processor cores, of open source RDBMS products, on average, equal or exceed the scalability of proprietary RDBMS products, on average, when run on the same operating system and hardware?

The benchmark scores are based on duration, so a lower score reflects higher performance. The scores were normalized to allow easy comparison. What follows is a description of the how the suite of benchmark case tests were constructed and used to collect the data that measured the relative performance and scalability of the selected RDBMS products.

Research Methods and Design

To compare performance of RDBMS products, the benchmark tests were run on each product using the same server with the same number of processor cores. The tests were repeated on the server with one and four processor cores. The first research question was answered by comparing the average performance of open source databases ($Y_{open-perf-1}$) against proprietary databases ($Y_{prop-perf-1}$) on the server with one processor core. The second research question was the same, except with four processor cores ($Y_{open-perf-4}$ and $Y_{prop-perf-4}$). To compare scalability, the results of the performance tests were compared to the number of processor cores and provide for a rate of increase in performance as processor cores are added. The third research question was answered by comparing the average scalability of open source databases ($Y_{open\text{-}scale}$) against proprietary databases ($Y_{prop\text{-}scale}$) by examining the average rate of increase in performance.

A benchmark case was created consisting of three parts, to simulate the different kinds of load a real-world application may incur. There are tests to reflect batch processing loads, online transaction processing, and online analytical processing. The case was run on the server using one processor core and again with four processor cores, to show how products scale up when processors are added to the server. The scalability of RDBMS products is particularly relevant with most computers adopting multi-core chip technologies in the near future (Spooner, 2005).

It is desirable to have the benchmark case model a realistic problem. The application tasks included in the benchmark case tests should represent the typical use cases. As a result, a data model was created that would describe the workings of a hypothetical chain of retail stores. It involves the creation of entities like customers and their addresses; club memberships; transactions and their contents; item ratings submitted by customers; stores and their inventories; departments and their discounts at various stores; individual items for sale; shipping charges and tax rates; and volume discounts. The entities were selected to create a simple model; a real application would go into more detail with many more entities and attributes. The model is described more clearly in Figure 1.

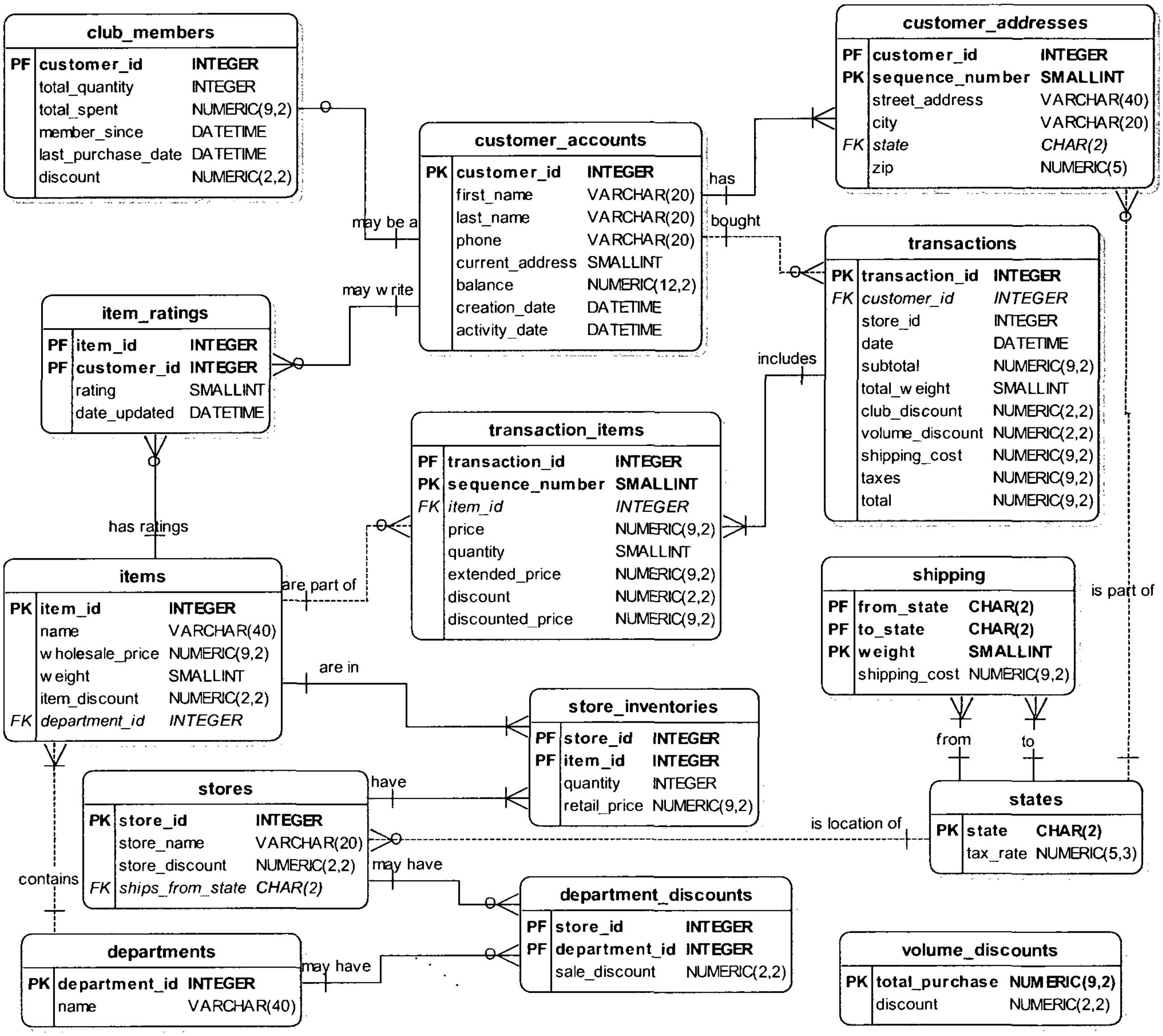

Figure 1. The database schema for the database benchmark case, including table names, field names, keys, data types, and relationships.

The data model includes several kinds of discounts and many relationships, so that the activities on the database required complex queries and that individual units of work involved several tables. Each of the database systems tested created the 14 tables described in the model.

The first part of the benchmark case measured batch load time by loading data from predetermined raw data files that were created before any of the benchmark case tests were run. The raw data files were the same for every database system tested; thus, the data only needed to be generated once. The test data was created using software written in Perl to create text files containing the data. Each file contained data for one table, with one line per record and column data separated by commas. In order to simplify the benchmark case test scripts, there are no commas within the actual data fields themselves.

The test data were the same for all databases and all benchmark case test runs to avoid variations in test data affecting the outcome of the performance measurements. If the data generated had been different for each database, it is possible that one RDBMS product might have received data that would have been easier to query and process, thus leading to an unfair advantage. To maintain a fair comparison, the same data set was used for all RDBMS products.

What follows is a description of the 14 tables involved in the test, and how the raw data was generated for the text files. Each of the tables is also described in the data model shown in Figure 1.

The *states* table serves as a list of valid states in the system. Each state also has a tax rate associated with it. The *state* column serves as a foreign key in the *stores*, *shipping*, and *customer_addresses* tables. To create the states table, a list of the 50 states was used and a tax rate was assigned to each state. The tax rate was randomly chosen for each state with values between 6.5% and 8.875% using the code listed in Appendix A. The states table will have 50 records.

The *departments* table serves as a list of valid departments. Each department has a *department_id* and a *name*. The department_id column serves as a foreign key in the *department_discounts* and *items* tables. The department names were randomly generated using the code listed in Appendix B. The departments table generation code randomly selected a name from each of two lists and combined them. The code also ensured that no name was repeated twice. The randomly generated names may not be very realistic but sufficed for the queries that were run as part of the benchmark. The departments table has 60 records.

The *stores* table contains a list of the stores used by the fictional company the benchmark case is modeling. Each store consists of a *store_id*, a *store_name*, a *store_discount*, and a *ships_from_state* column. The store_id column also serves as a foreign key in the department_discounts and *store_inventories* tables. The stores data file was created using the code listed in Appendix C. The store_name was generated by randomly picking one word each from two lists, and appending that with a number designating the instance of that pair, for example, *Mega Mart 1, Mega Mart 2*, and so on. The store_discount was randomly assigned with an 80% chance for 0% discount and a 5% chance for each of 5%, 10%, 15%, and 20% discounts. The ships_from_state was chosen randomly from a list of 50 states. The stores table has 250 records.

The department_discounts table contains a list of the sale discounts currently present in the various departments at the various stores. Each department discount consists of a *store_id*, a *department_id*, and the *sale_discount*. The first two fields combine to create a composite primary key. Values were randomly assigned for each combination of store and department. To simulate the effect of not every store having every department, each combination had only a 20% chance of existing in the table. For those combinations of store and department that exist, the amount of the discount had a 50% chance of being 0%, and a 10% chance each of being one of 5%, 10%, 20%, 30%, or 40%. The data file was created using the code listed in Appendix D. The department_discounts table has about 3000 rows.

The *shipping* table details the cost of shipping an item from one state to another, depending on its weight. There are four columns, which are *from_state*, *to_state*, *weight*, and *shipping_cost*. The first three fields combine to create a composite primary key. The code to generate the data for the shipping table is listed in Appendix E. There are ten weight categories and a base cost associated with each. For each combination of from_state and to_state, a random factor was selected from 1.1, 1.2, 1.3, 1.4, 1.5, and 1.6. The random factor was multiplied by the base shipping cost for each weight category for that state pair. Although the random distribution does not match a realistic shipping schedule, it sufficed for the purposes of the benchmark. There are 25,000 records in the shipping table.

The items table describes the various items available for sales in the various stores. Each item consists of an *item_id*, *name*, *wholesale_price*, *weight*, *item_discount*, and *department_id*. The item_id also serves as a foreign key in the *item_ratings*, *transaction_items*, and store_inventories tables. The code to generate the items and their information is listed in Appendix F. The *name* of each item was created by randomly picking between two and five words from seven lists and concatenating them together. As a result, some of the names may appear to be nonsense when read by the human eye (for example, *Felt Slow Cool Cool*, *Pants Yellow Fast*, *Premium Premium Socks*, and so on).

The randomly generated names served well for the benchmark testing and allowed for randomly searching on various keywords. The data generation program limits the length of any name to 40 characters, truncating to 40 any names that are longer than that. The wholesale_price was randomly assigned a value between 10.00 and 610.00. The *weight* was randomly assigned a number between 1 and 20. For item_discount, there was an 80% chance of 0% discount and a 5% chance each of a 5%, 10%, 15%, or 20% discount. The item was also randomly assigned to one of the 60 department_id values. The items table had approximately 25,000 records.

Raw data for customer names was acquired from the U.S. Census Bureau (2005), which listed the most common surnames, male first names, and female first names. Along with each name it listed the relative frequency at which the names occurred in the population. The census data files were read into a spreadsheet and adjusted to create a cumulative distribution function and converted in a format that could be read by the Perl programs for the creation of customer names. The resulting data files *male.txt*, *female.txt*, and *lastnames.txt* were created by the spreadsheet. Similarly, a list of population estimates for U.S. cities with a population of 10,000 or more was acquired from the U.S. Census Bureau (2000). The census data was also read into a spreadsheet and converted into a cumulative distribution function. The data file *cities.txt* was derived from the spreadsheet. Using the census data allows the customer data to achieve a more realistic appearance.

The *customers* table has information about the customers of the fictional company the benchmark case is modeling. Each customer record has a *customer_id*, *first_name*, *last_name*, *phone*, *current_address*, *balance*, *creation_date*, and *activity_date*. The code to generate the data for the customers table and the *customer_addresses* table is listed in Appendix G. It requires data files that were generated as described in the previuos paragraph. The customer_id serves as a foreign key in the customer_addresses, *transactions*, *club_members*, and item_ratings tables. For the first_name, the gender was determined randomly with a female name appearing with a 52% chance. Using a random number and the cumulative distribution function from the male and female first names, a first name was selected. A last_name was chosen in a similar manner. The phone number was generated randomly. Each customer can have several addresses in the customer_addresses table, and the current_address field maps to one of those addresses. The customer's balance is 0.00 95% of the time, and a random value between 0.00 and 2000.00 otherwise. For the creation_date, a date was randomly selected between January 1, 2003 and December 31, 2007. The creation_date indicates when the customer account was created. For the purposes of the benchmark, the time period was hardcoded to the day after the five-year period mentioned previously, which was January 1, 2008. The activity_date indicates the last time the customer accessed the system and was given a date randomly chosen from the last six months of the same period the creation_date was chosen from, so it was a value between July 1, 2007 and December 31, 2007. If the creation_date was after July 1, 2007 then the activity_date was set to a date randomly chosen between the creation_date and December 31, 2007. There are about 1,000,000 records in the customers table.

The customer_addresses table contains addresses belonging to customers in the customers table. Each record consists of a *customer_id*, *sequence_number*, *street_address*, *city*, *state*, and *zip*. The code that generated the data for the *customers* table also created the data for the customer_addresses table at the same time. The customer_id and sequence_number form a composite primary key for the table, with the sequence_number unique for a given customer_id only, starting with 1 for each customer. The street_address was randomly generated by picking a house number between 1 and 10,000, a street number between 1 and 150, one of eight directions, the word *Road*, *Street*, *Avenue*, or *Boulevard.* The city and state were randomly selected from the cumulative distribution file created for cities earlier. The zip was a randomly generated five-digit number. Each customer was randomly determined to have between 1 and 5 addresses. The generated data resulted in approximately 3,000,000 records in the customer_addresses table.

The item_ratings table contains ratings by customers of various items the company sells. Each record has an *item_id*, *customer_id*, *rating*, and *date_updated.* The item_id and customer_id form a composite primary key for the table. The code to generate the data for the item_ratings table is listed in Appendix H. For every item, there was an 11% chance of having 0 ratings, a 78% chance of having between 1 and 10 ratings, an 11% of having between 1 and 100 ratings, and a 1% chance of having between 1 and 1000 ratings. Using a nonuniform distribution provides an interesting spread of items and their ratings. The date_updated was randomly determined to be between the customer's creation_date and the end date of the model, December 31, 2007. There were approximately 360,000 rows in the item_ratings table.

The store_inventories table describes the inventory of items in each store. Each record consists of a *store_id*, *item_id*, *quantity*, and *retail_price*. The store_id and item_id make up the composite primary key. The code to generate the store_inventories table is listed in Appendix I. For the quantity, there was a 40% chance of there being 0 quantity for a given item, otherwise a random amount between 1 and 6 items. If the quantity is 0, then there was no record for that item and store. The retail_price was randomly set to the wholesale price plus a markup between 5% and 40%. The generated data had approximately 3,750,000 records in the store_inventories table.

The *volume_discounts* table provides a schedule of customer discounts based on the total amount of a purchase. The data was generated by the code in Appendix J. Each record has two fields, *total_purchase* and *discount.* The total purchase amount starts with 100, 200, 400 and doubles on the way up to 102,400. The discount starts with 0%, 1%, 2% and works its way up to 10%. The volume_discounts table has 11 rows.

The transactions and transaction_items tables require some of the data from the club_members table, but the rest of the club_members table cannot be generated until information about the transactions is known. First, an interim version of the club_members table was created that has only two fields, the customer_id and the *discount.* The code to generate the interim table is listed in Appendix K. For each customer account, there was a 15% chance of it being a club member. Club membership has three levels of discount: a 75% chance of a 5% discount, a 20% chance of a 10% discount, and a 5% chance of a 15% discount. The interim file had about 150,000 records.

The next program uses the output of the previous program, the interim file. It generates the transactions and transaction_items table simultaneously. Once the transaction data was completely generated, the final version of the club_members table was generated, using sums that were calculated on the fly while generating transactions.

The program to generate the transaction records and final club_members data is listed in Appendix L. About 15,000,000 transaction records were created; the number was random because it started with a loop over 30,000,000, choosing a customer account at random, and only generating a transaction if the randomly generated transaction date is greater than the customer's creation date. Each row in the transactions table consists of the fields *transaction_id*, *customer_id*, *store_id*, *date*, *subtotal*, *total_weight*, *club_discount*, *volume_discount*, *shipping_cost*, *taxes*, and *total.* The transaction_id and customer_id make up the composite primary key for the table. The store_id was randomly selected. Each transaction had a 75% chance of having between 1 and 3 transaction items, and a 25% chance of having between 1 and 19 transaction items. The random distribution was selected to result in approximately 4 items per transaction. The generated data resulted in about 60,000,000 rows for the *transaction_items* table. The transaction_items table has the fields *transaction_id*, *sequence_number*, *item_id*, *price*, *quantity*, *extended_price*, *discount*, and *discounted_price*. The transaction_id and sequence_number make up the composite primary key for the table. The sequence_number identifies individual items for a transaction. The item_id was randomly selected from the items table. The price was from the items table as well, and marked up by a random amount of -20% to +40%. The quantity was set to 1 90% of the time and otherwise was a random number between 1 and 4. The weight was taken from the items table, multiplied by the quantity, and added to a sum (total_weight) for each transaction. The extended_price was the price times the quantity. 50% of the time the discount was set to 0%, and a random value between 1% and 40% otherwise. The discounted_price was then computed from the extended_price and the discount. The sum of the discounted_price values for a given transaction became that transaction's subtotal. The volume_discount and club_discount were computed for the transaction based on the customer's club membership (if any) and the subtotal. The shipping_cost was computed based on values in the shipping table, based on the store's state, the customer's state, and the total_weight. The tax_rate was taken from the states table, based on the customer's state. Finally the total was computed for the transaction based on the other values.

The same program used to create the data for the transation and transaction_items tables also generated values for the club_members table by summing values while computing data for each transaction. The fields in the final club_members table include *customer_id*, *total_quantity*, *total_spent*, *member_since*, *last_purchase_date*, and *discount*. The customer_id and discount were copied from the interim club member table. The total_quantity and total_spent were summed from each transaction as they were generated. The member_since date was a randomly generated date between January 1, 2003 and December 31, 2007. If the randomly generated date was less than the customer creation_date, then it was replaced with the customer creation_date. The last_purchase_date was the date of that customer's most recent transaction, or null if the customer had no transactions. The club_members file had about 150,000 records, the same number as the interim club members file.

The test data was generated only once, using the test data generation programs. A copy of the original 14 data files was used in each run of the benchmark testing of each database product. Every RDBMS product used the same data, which helped to provide a fair comparison between databases products. The size of the data generated was designed to be significant enough to require a nontrivial amount work for each database product. In particular, the resulting database size was larger than the amount of memory available on the system, which meant that the database application could not simply cache all of the data into memory. Systems that arrange data on disk more intelligently and have smarter caching and querying algorithms performed better than others that did not.

The benchmark case was divided into three parts: loading the data, performing transactions, and generating reports. Loading the data models a batch data feed from an external system being used to load the data into the database. For each database, a script read the data from the raw text files and inserted the values into the tables. The 14 tables were loaded in parallel. The second phase of the benchmark case involved running transactions that simulate typical business activities. Typical transactions involved inserting, updating, and deleting data. A master driver script ran 20 clients in parallel, and each executed a series of 500 randomly selected transactions. When all of the transactions were completed, the second phase was finished. Finally in the third phase, a collection of reports were run. All of the reports ran in parallel. Some of the reports involved complex queries that look at large amounts of data. When all of the reports were finished, the third phase was completed.

The script to load the data truncated all of the tables and started 14 child processes, one for each test file and database table associated with the test file. Each child process loaded one table by reading data from a copy of the original raw text file and executing insert statements into the database. Every RDBMS product used the same test data. Executing insert statements was the most portable solution, but still required the date fields to be translated into a format acceptable by the database system. Some database products, like Sybase, support a bulk copy feature that allows for faster loading if the raw text file is prepared in a certain format; but for the purposes of the benchmark case the more general method of simply using insert statements was used. The benchmark load script can be seen in Appendix M. The raw text files containing the data were created in a comma-separated-values format, with the fields in the order described in the data model shown in Figure 1. The data load was repeated several times, using an empty database each time. Batch load times were recorded for further analysis. After the last batch load, a copy of the database was made, which was used to restore the database image for each transactional processing test run.

The second part of the benchmark case measured transactional processing. For the transactions, each of the 20 clients will perform 500 random transactions. Transactions were divided into three types, which were called very common transactions, common transactions, and rare transactions. The names "very common", "common", and "rare" were selected to easily represent the relative importance of each transaction. Each client will perform a very common transaction 60% of the time, a common transaction 30% of the time, and a rare transaction 10% of time. The percentages for each of the three types of transactions were selected based on the author's personal experience with production systems. For the purposes of the benchmark case, the selected the percentages reflect that some transactions happen more frequently than others. There are several of each type of transaction, and the particular transaction was chosen randomly. For the purposes of the transactions, the current date was assumed to be January 1, 2008, because the generated data contained dates between January 1, 2003 and December 31, 2007.

The following is a list of the very common transactions:

Create a customer and address. A new record is inserted into the customers and customer_addresses tables.

Update a customer balance. The balance is read from a randomly chosen customer and updates the value in the customers table, subtracting 100 from it.

Customer rates an item. A randomly selected customer and item are chosen. If an item_rating record already exists for the selected customer and item, then the *rating* is updated to a random value between 1 and 5 and the date_updated is set to January 1, 2008. If a record does not exist, then a new one is created.

Customer purchases items. A random number of items and quanitities is selected following the same chances that were used in the generation of the data. A store is randomly chosen and items are randomly selected from the store_inventories table. A randomly selected customer is chosen and records are inserted into the transactions and transaction_items tables based on the data fetched from the transactions, transaction_items, volume_discounts, club_members, shipping, and states tables, which contain the necessary information to complete the transaction. The club_members table is also updated to reflect new values for total_quantity, total_spent, and last_purchase_date.

Add to or remove from a store's inventory. A randomly selected store and item are selected. If the store has no inventory in that item, a new record is inserted into the store_inventories table. If a record exists, then the record is deleted.

Remove an item from all stores. An item is randomly selected. The store_inventories table is updated with zero quantity for all stores carrying the item. If it does not exist, then an item is created.

The common transactions are:

Update customer phone. The phone field of a randomly selected record in the customers table is changed.

Update customer address. The street_address, city, state, and zip of a randomly selected record in the customer_addresses table is changed.

Add new address. A new record is inserted into the customer_addresses table for a random customer.

Change current address. The highest sequence_number value from the customer_addresses table is selected for a random customer and the current_address field of the customers table is randomly set to a value less than or equal to the highest sequence number.

Customer joins club. A randomly selected customer is chosen. If the customer is not in the club_members table, a new record is inserted with a discount of 5%, a member_since of January 1, 2008, 0 for total_spent and total_quantity, and null for last_purchase_date. If the customer is already a member, then their discount will be increased by 5%, unless it is already 15%, in which case it will be lowered to 5%.

A new item is created. New information is randomly generated for the item, and values are inserted into the items table. In addition, 20% of stores will get inventory of the item and records will be inserted into the store_inventory table. The quantities and retail_prices are randomly determined using a method similar to what was used in the data generation program.

An item will be updated. The wholesale_price and item_discount fields for one record in the *items* table are updated to a new random amount. The store_inventory table is updated with new retail_price values for that item as well.

The rare transactions are:

A new store is created. Values are randomly generated for the new record in the stores table. New records are also inserted into store_inventory and department_discounts tables as well, using methods simliar to that used for the generation of the initial data, except with far fewer items added to the store's inventory.

A store is closed. A store number is randomly selected. The record will be deleted from the stores table and related records will be deleted from the store_inventory and department_discounts tables. If the randomly selected store does not exist, then the transaction creates a new store instead.

A store's discount is updated. A randomly chosen store has its record updated in the stores table. The retail_price is also updated in the store_inventories table as well for that store.

Add or remove a department discount. A store and depart are randomly selected. If the pair has a record in the department_discounts table, then it will be deleted. Otherwise, a randomly selected sale_discount is selected and a new record is inserted into the table.

Update the volume discounts. Every total_purchase value in the volume_discounts table is increased by 10%, or every total_purchase value is decreased by 10%.

Update shipping costs. Every shipping_cost value in the *shipping* table is increased by 10%, or every shipping_cost value is decreased by 10%.

Change the club member discounts. Either increase all of the discount values in the club_members table by 10%, or decrease them all by 10%.

The program driving the transactions consists of two parts. The first part has database-independent code, and controls the random value generation and basic SQL statements. The second part has the database-dependent code, which is unique for every database. The benchmark transaction script is listed in Appendix N. Efforts were made to minimize the amount of database-dependent code, to simplify the programming and provide for a fair comparison between database systems.

The program driving the transactional tests measured how long it takes to complete all of the 20 processes to run their 500 transactions. The measured time was recorded as one sample. The database was then restored to a copy of where it was before the transactional processing started, and the transactional processing test was then repeated. The transactional processing test was repeated several times, with each time being recorded for further analysis.

The third part of the benchmark case involve report generation. The reports model possible queries used in decision support. In the report generation phase of the benchmark case, the reports were run in parallel.

The reports are:

Store Profits 1. List the stores in order by total profits for the day. Profit is retail price minus wholesale price. Also list the number of items sold, the total weight, and the percentage of club members making the purchases.

Store Profits 2. Run the Store Profits 1 report except with a time frame of the past month.

*Store Profits 3.*Run the Store Profits 1 report except with a time frame of the past year.

State Items 1. Report by state the total number of items shipped for the past day. Include the total shipping costs, and total weight.

State Items 2. Run the State Items 1 report except with a time frame of the past month.

State Items 3. Run the State Items 1 report except with a time frame of the past year.

Department Revenues 1. Report by department showing the departments with the highest revenues for the past day. Also show number of items and number of stores.

Department Revenues 2. Run the Department Revenues 1 report except with a time frame of the past month.

Department Revenues 3. Run the Department Revenues 1 report except with a time frame of the past year.

Most Popular Items. List the 100 most popular items by quantity for the past year.

Most Profitable Items. List the 100 most profitable items for the past year, using total profits generated.

State Customers 1. List the states ordered by number of customers for the past day. Use the current addresses of the customers.

State Customers 2. Run the State Customers 1 report except with a time frame of the past month.

State Customers 3. Run the State Customers 1 reprot except with a time frame of the past year.

City Customers 1. List the cities ordered by number of customers for the past day. Show only the top 100 cities.

City Customers 2. Run the City Customers 1 report except with a time frame of the past month.

City Customers 3. Run the City Customers 1 report except with a timeframe of the past year.

Top Customers 1. List the top 100 customers for the past day. Use the total spent by each customer.

Top Customers 2. Repeat the Top Customers 1 report except with a timeframe of the past month.

Top Customers 3. Repeat the Top Customers 1 report except with a timeframe of the past year.

The generated data will assume a historical period between January 1, 2003 and December 31, 2007. The transaction processing tests assume a current date of January 1, 2008. For the report generation, *past day* means January 1, 2008, *past month* means the period from December 1, 2007 to January 1, 2008, and *past year* means the period from January 1, 2007 to January 1, 2008.

Before the report generation starts, the database were halted and restarted, to flush out any data values from memory caches. The system then measured the time it took to complete all of the reports, which ran in parallel. The time measured started when the report generation benchmark case test started, and ended when the last report completed. The database was then halted and restarted again for the next run of the report generation test. The system halts and restarts occured outside the measurement time, and were used to ensure the database started from the same initial condition each time. The time measurements were collected and used for further analysis. The benchmark report generation script is listed in Appendix O.

The benchmark case consists of one set of batch load tests, transaction processing tests, and report generation tests. Each case had several run times for each part. Once the case completed for a given database product running with one processor, the entire process was repeated with the server running with four processors. When that was completed, the testing then moved on to the next database product being reviewed. The testing then continued until all of the database systems had been measured.

Participants

The benchmark case is a non-experiment measuring the performance of software applications, involving no human participants. The benchmark case was run against the most popular proprietary and open source RDBMS products, and time measurements were taken for each benchmark case test run. As there are no treatments, measurements cannot be considered an experiment, so this qualifies as a non-experiment. The subjects of the measurements are the RDBMS products, so no human and no animal participants were used.

Three proprietary and three open source RDBMS products were examined. These six products have a combined market share of over 95% on the Linux platform (Chen, 2002). The results of this research are thus intended to generalize to the whole population of RDBMS products available on the Linux platform.

Materials/Instruments

The benchmark case tests were performed on a PC running Linux. The hardware consisted of an Intel quad-core processor, one gigabyte of memory, and two 320 gigabyte SATA drives in a mirrored configuration. The benchmark case was written in Perl, and ran on the same host as the database, eliminating the need for any network considerations. The Perl DBI library was used to access the database, which allowed for most of the code to be written in a database-independent manner.

To maintain control in the test environment, all of the tests were performed on the same hardware and operating system. For each benchmark case, the same server was used, with the same operating system. Only processes belonging to the operating system, the RDBMS, and the benchmark case tests were run on the server. The server was restarted before each test to ensure that each test had a similar operating environment. No other applications were run on the server during the tests

The test environment was a PC running Linux. The choice of hardware platform reflected the changing economics of servers today. Hence, a server with multiple cores was chosen, but due to costs constraints the hardware purchased turned out to be a quad-core server with a 2.4 GHz Intel Core 2 Quad® Q6600 processor running Linux.

Linux was used because it is rapidly being adopted by many firms for server applications (Silwa, 2005), and most database software vendors have a Linux version available, including Oracle® (Songini, 2003) and IBM® (Campbell, 2002). Unfortunately, running the tests on Linux excluded Microsoft® SQL Server from being tested, as it only runs on Windows®. Comparing the performance of different RDBMS products on different platforms is difficult, because one cannot tell which part of the performance is attributable to the RDBMS and which is attributable to the platform (Apicella, 2000). The proprietary RDBMS products are from other vendors, but their names are not included in the results because some proprietary RDBMS products do not allow benchmark tests of their products to be published without their written consent (Bruckler, 2005). The proprietary RDBMS products Oracle® 10g, Sybase® Adaptive

Server Enterprise 15, and IBM® DB2 9.5 were randomly assigned the names Database A, Database B, and Database C. The open source RDBMS products tested were MySQL®, PostgreSQL, and Firebird®. Alternative RDBMS products like Sleepycat, One$DB, and other similar products were not used in the benchmark case, because Sleepycat and One$DB are generally considered for embedded use and not as standalone database servers.

Operational Definition of Variables

This section describes all of the independent and dependent variables that are inputs to the computation of the benchmark. There are four independent variables: the RDBMS product, the number of processor cores, the benchmark case test, and the benchmark case test run number. For each of these four values test durations were taken. The rest of the dependent variables are constructed upon the test durations, which were necessary for the analysis.

RDBMS Product: Independent variable (X_d). The specific database product being tested. Some database products were open source and some were proprietary.

Number of Processor Cores: Independent variable (X_p). The number of processor cores used by the database system during the test, either X_p=1 or X_p=4.

Benchmark Case Test: Independent variable (X_t). There are three tests that make up the benchmark case. X_t=1 represents the batch load test. X_t=2 represents the transaction processing test. X_t=3 represents the report generation test.

Benchmark Case Test Run: Independent variable (X_r). Each test will be run 10 times, so X_r will take on the values 1 through 10.

Test Duration: Dependent variable ($Y_{duration\text{-}d,p,t,r}$). The duration of a benchmark case test run for a given database product, number of processor cores, and benchmark case test. The benchmark case test programs take clock measurements at the start and finish of each test. The duration is calculated as the difference of these clock values.

Normalized Average Duration: Dependent variable ($Y_{navg\text{-}d,p,t}$). The average duration of the benchmark case test runs for a given database product, number of processor cores, and benchmark case test, normalized by dividing it by the equivalent score for MySQL, and multiplying by 100.

Performance: Dependent variable ($Y_{perf\text{-}d,p}$). The performance of a given database product and number of processors, using a weighted average of the normalized average durations. The formula to be used is $Y_{\text{perf-d,p}} = (0.15)Y_{\text{navg-d,p,1}} + (0.55)Y_{\text{navg-d,p,2}} + (0.3)Y_{\text{navg-d,p,3}}$.

Scalability: Dependent variable ($Y_{scale\text{-}d}$). The scalability is the rate of improvement of performance for a given database product when going from one processor core to four processor cores, as given by the formula $Y_{\text{scale-d}} = Y_{\text{perf-d,4}} / Y_{\text{perf-d,1}}$.

Performance of Proprietary RDBMS Products with One Processor Core: Dependent variable ($Y_{prop\text{-}perf\text{-}1}$). The average of the values $Y_{\text{perf-d,1}}$ for those database products that are proprietary.

Performance of Proprietary RDBMS Products with Four Processor Cores: Dependent variable ($Y_{prop\text{-}perf\text{-}4}$). The average of the values $Y_{\text{perf-d,4}}$ for those database products that are proprietary.

Scalability of Proprietary RDBMS Products: Dependent variable ($Y_{prop\text{-}scale}$). The average of the values $Y_{\text{scale-d}}$ for those database products that are proprietary.

Performance of Open Source RDBMS Products with One Processor Core: Dependent variable ($Y_{open\text{-}perf\text{-}1}$). The average of the values $Y_{\text{perf-d},1}$ for those database products that are open source.

Performance of Open Source RDBMS Products with Four Processor Cores: Dependent variable ($Y_{open\text{-}perf\text{-}4}$). The average of the values $Y_{\text{perf-d},4}$ for those database products that are open source.

Scalability of Open Source RDBMS Products: Dependent variable ($Y_{open\text{-}scale}$). The average of the values $Y_{\text{scale-d}}$ for those database products that are open source.

Table 1

Construct Variables

Construct Variable	Definition	Possible Values
$Y_{duration\text{-}d,p,t,r}$	Duration, in minutes, for RDBMS X_d to run the benchmark case test X_t with number of processor cores X_p during run X_r.	> 0.
$Y_{navg\text{-}d,p,t}$	The normalized average duration for RDBMS X_d, test X_t, and number of processor cores X_p.	> 0. Equals 100.000 for MySQL.
$Y_{perf\text{-}d,p}$	Performance of RDBMS X_d with number of processor cores X_p.	> 0, between min and max of $Y_{navg\text{-}d,p,t}$ for a given X_d and X_p.
$Y_{scale\text{-}d}$	Scalability of RDBMS Xd, taken as $Y_{perf\text{-}d,4}/Y_{perf\text{-}d,1}$	> 0, most likely between 0.250 and 1.000.
$Y_{prop\text{-}perf\text{-}1}$	Average performance of proprietary RDBMS products with one processor core, taken as the average of $Y_{perf\text{-}d,1}$ where X_d is in the set of all proprietary RDBMS products tested.	> 0, between min and max of $Y_{perf\text{-}d,1}$ for X_d in the set of all proprietary RDBMS products tested.
$Y_{prop\text{-}perf\text{-}4}$	Average performance of proprietary RDBMS products with four processor cores, taken as the average of $Y_{perf\text{-}d,4}$ where X_d is in the set	> 0, between min and max of $Y_{perf\text{-}d,4}$ for X_d in the set of all proprietary

	of all proprietary RDBMS products tested.	RDBMS products tested.
$Y_{\text{prop-scale}}$	Average scalability of proprietary RDBMS products, taken as the average of $Y_{\text{scale-d}}$ where X_d is in the set of all proprietary RDBMS products tested.	>0, between min and max of $Y_{\text{scale-d}}$ for X_d in the set of all proprietary RDBMS products tested.
$Y_{\text{open-perf-1}}$	Average performance of open source RDBMS products with one processor core, taken as the average of $Y_{\text{perf-d,1}}$ where X_d is in the set of all open source RDBMS products tested.	> 0, between min and max of $Y_{\text{perf-d,1}}$ for X_d in the set of all open source RDBMS products tested.
$Y_{\text{open-perf-4}}$	Average performance of open source RDBMS products with four processor cores, taken as the average of $Y_{\text{perf-d,4}}$ where X_d is in the set of all open source RDBMS products tested.	> 0, between min and max of $Y_{\text{perf-d,4}}$ for X_d in the set of all open source RDBMS products tested.
$Y_{\text{open-scale}}$	Average scalability of open source RDBMS products, taken as the average of $Y_{\text{scale-d}}$ where X_d is in the set of all open source RDBMS products tested.	>0, between min and max of $Y_{\text{scale-d}}$ for X_d in the set of all open source RDBMS products tested.

Data Collection, Processing, and Analysis

The results of the testing with the proprietary products using one processor core were combined into one score that represents the proprietary RDBMS products.

Likewise, the scores for the open source RDBMS products using one processor core were combined into one score representing open source products. The combined scores were used as the basis for comparison in answering the primary research question. Similar testing was repeated using four processor cores, so that open source and proprietary RDBMS products could be compared in a multi-core environment. Finally, the measure of scale-up from one processor core to four processor cores indicated the scalability of open source and proprietary RDBMS products.

The dependent variables measured were the completion times required for a selected RDBMS (independent variable X_d) and for an assigned number of processor cores (independent variable X_p) to perform each benchmark case test (independent varaible X_t) during a benchmark case test run (independent variable X_r). The benchmark case was run for each database product X_d, once with one processor core (X_p=1) and once with four processor cores (X_p=4). The benchmark case consists of three tests: the batch load of the data (X_t=1), processing transactional updates (X_t=2), and analytical query processing for report generation (X_t=3). Each test X_t was run 10 times (X_r takes on values 1 through 10). The duration of a test run for a given X_d, X_p, X_t, and X_r was recorded as the dependent variable $Y_{duration-d,p,t,r}$.

To make sure that the environment was the same at the start of each test, the host server was rebooted before each benchmark test, to ensure that no data was cached in a filesystem. Additionally, the data to be loaded into the database was the same for each test. At the start of the database load, the database was empty. The database was then loaded with data from a predetermined data file that was the same for all benchmark case tests. Once the load was completed, the online transaction processing began. The online transaction processing consisted of a predetermined number of random transactions performed in parallel by a predetermined number of drivers. Once the last online transaction had been processed, the analytical processing began. The analytical processing consisted of a predetermined, non-random number of reports which were performed in parallel. When the last report completed, the benchmark case test recorded the total duration of the test and the test ended.

Once all of the raw data for $Y_{duration-d,p,t,r}$ was collected, it was combined into intermediate construct variables and then into the final construct variables representing the performance and scalability of open source and proprietary RDBMS products.

The first intermediate construct variable is the normalized average duration, $Y_{navg-d,p,t}$, which is computed by taking the average duration of the benchmark case test runs for a given database product X_d, number of processor cores X_p, and benchmark case test X_t, normalizing it by dividing it by the equivalent score for MySQL, and multiplying by 100. Thus, MySQL's numbers were 100.000 for each test, and the other database products were measured relative to MySQL, with faster products scoring less and slower products scoring more, because the values were derived from duration. Data was recorded to three decimal places of precision, which was sufficient to indicate differences between scores.

Normalized average duration was used because each of the three tests require different amounts of time to complete, and the next intermediate construct variable, performance, is a weighted average of the normalized average durations. The formula for the performance ($Y_{perf-d,p}$) of a given database product X_d and number of processor cores X_p is given as $Y_{perf-d,p} = (0.15)Y_{navg-d,p,1} + (0.55)Y_{navg-d,p,2} + (0.3)Y_{navg-d,p,3}$. $Y_{perf-d,p}$ weights the batch processing test with 15% of the final value, the transaction processing test with

55% of the final value, and the report generation test with 30% of the final value. The percentage weights were selected because they are similar in relative importance to the three types of load frequently used in production applications. Because each test is normalized to MySQL, the $Y_{navg\text{-}d,p,t}$ scores can be added together and still maintain their weights. The scalability ($Y_{scale\text{-}d}$) of a given database product X_d is computed as the ratio of the performance with four processor cores divided by the performance with one processor core, as given by the formula $Y_{scale\text{-}d} = Y_{perf\text{-}d,4} / Y_{perf\text{-}d,1}$.

From the performance ($Y_{perf\text{-}d,p}$) and scalability ($Y_{scale\text{-}d}$) scores the values necessary to answer the research questions were computed. The performance of proprietary RDBMS products with one processor core, $Y_{prop\text{-}perf\text{-}1}$, was the average of the values $Y_{perf\text{-}d,1}$ for those database products that are proprietary. Similarly the performance of open source RDBMS products with one processor core, $Y_{open\text{-}perf\text{-}1}$, was the average of the values $Y_{perf\text{-}d,1}$ for those database products that are open source. A statistical analysis of $Y_{open\text{-}perf\text{-}1}$ and $Y_{prop\text{-}perf\text{-}d,1}$ using the *t* statistic answered the primary research question.

Similarly, the performance of proprietary RDBMS products with four processor cores, $Y_{prop\text{-}perf\text{-}4}$, was the average of the values $Y_{perf\text{-}d,4}$ for those database products that are proprietary and $Y_{open\text{-}perf\text{-}4}$ was the average of the values $Y_{perf\text{-}d,4}$ for those database products that are open source. A statistical analysis of $Y_{open\text{-}perf\text{-}4}$ and $Y_{prop\text{-}perf\text{-}4}$ using the *t* statistic answered the second research question.

Finally, scalability of proprietary RDBMS products, $Y_{prop\text{-}scale}$, was the average of the values $Y_{scale\text{-}d}$ for those database products that are proprietary. $Y_{open\text{-}scale}$ was the average of the values $Y_{scale\text{-}d}$ for those database products that are open source. A statistical analysis of $Y_{open-scale}$ and $Y_{prop-scale}$ using the *t* statistic answered the third research question.

For the primary research question, the number of processor cores, X_p, was held at 1. Thus there are only three independent variables, the RDBMS, X_d, the test, X_t, and the test run, X_r. The three variables X_d, X_t, and X_r can function as one variable, as any given test run X_r will be for a specific RDBMS X_d and test X_t. As a result, for the primary research question, there is no need for ANOVA to test for interactions among the independent variables. The resulting derived variables, $Y_{prop-perf-1}$ and $Y_{open-perf-1}$, required an analysis of two random variables, using techniques described in chapter 8 of Aczel and Sounderpandian (2002). Using a 95% confidence interval (α=0.05) it can be determined whether or not $Y_{open-perf-1}$ >= $Y_{prop-perf-1}$. In particular, the two means are compared using the *t* statistic. The populations are based on independent samples and thus expected to be normally distributed. As different RDBMS products may vary, it is not expected that their variances will be equal, and the homogeneity of variances were not be tested. Each sample is independent. The first research question, comparing the performance of proprietary and open source database products with one processor, is thus equivalent to the hypothesis that $Y_{prop-perf-1}$ - $Y_{open-perf-1}$ <= 0, because a lower score considered better.

To address the second research question regarding performance with four processors, similar techniques are used, but with X_p held at 4. Thus the second research question is equivalent to the hypothesis that $Y_{prop-perf-4}$ - $Y_{open-perf-4}$ <= 0.

When examining the third research question regarding scalability, the scalability scores for proprietary ($Y_{prop-scale}$) and open source ($Y_{open-scale}$) RDBMS products are computed. An interesting problem occured when computing the variance of a ratio of random variables due to the lack of closed form solutions. However, an estimate for the variance is available using the technique described by Kalton (1983), where the variance of a ratio of independent normal random variables A and B is given as $(1/\mu_B)$ times $[\sigma_A^2+(\mu_A/\mu_B)^2\sigma_B^2]^{1/2}$. Scalability of a database is measured by comparing the performance with four processor cores to the performance of one processor core, $Y_{scale\text{-}d} = Y_{perf\text{-}d,4} / Y_{perf\text{-}d,1}$. An analysis of two random values was performed to determine, at the 95% confidence level, if $Y_{open\text{-}scale} <= Y_{prop\text{-}scale}$. The *t* statistic was used to answer the third research questions using the hypothesis that $Y_{prop\text{-}scale} - Y_{open\text{-}scale} <= 0$. A lower value represents greater scalability.

There were some assumptions made with the benchmark case tests. First, given the same starting conditions, computers will generally tend to perform a task in the same amount of time. The batch load and report generation tests, in particular, having no random element, tend to have low variances in run time. The transaction processing tests have a random element and thus have more variance. The variance of the transaction processing test is reduced by the large number of transactions performed during the test, so variances were expected to be low, and thus only a small number of runs were made for each benchmark case test.

The benchmark case was assumed to be runnable in a reasonable amount of time. Given that there were six database systems to test, with two amounts of processor cores, and 10 runs per benchmark case test, each test was run 120 times. Since there were three different tests (batch load, transaction processing, and report generation), there were a total of 360 runs. In order to complete the research in a timely manner, the benchmark cast test runs needed to be able to complete in a relatively short amount of time. Thus early in the research phase the size of the test data and the number of transactions in the transaction processing test were adjusted before the official scoring benchmark cases were run.

It is further assumed that a benchmark case can be adjusted to suit an individual's own preferences regarding the valuation of the weights of the three benchmark test scores when measuring performance. A technology manager who does more transaction work may prefer, for example, to have the transaction processing normalized average be weighted at 80% of the performance score. The percentages used in this research were chosen to fit the author's own observations of real production systems.

Before analyzing the results of the experimental data, some comments need to be made as to the selection of the sample size for the experiments, which were selected after doing a power analysis of the test. Increasing the sample size while holding α and effect size constant will increase the power of the experiment. Increasing the sample size also reduces the variance. Theoretically, if one could increase the sample size to include the entire population, then variance would be reduced to zero; however, it would no longer be an experiment at that point. Realistically, there are cost constraints that limit the sample size to a small fraction of the population in most cases.

In Dyck (2002) and Strandell (2003), the benchmark methodology used involve running a benchmark test until a steady state was achieved; thus, each benchmark test was only run once. They did not treat the benchmark testing like an experiment. They also did not make any judgments as to whether the differences in the scores were statistically significant. As a result, a power analysis of their benchmark testing would be meaningless.

Using an online power analysis tool from Lenth (2006), the number of necessary runs can be estimated. One of Lenth's tools examines confidence interval for one mean. Using a sample size of 10, a confidence interval of 95% ($\alpha = 0.05$), and estimating the variance to be 0.05, the margin of error will be 0.03577. It is not possible to know the variance ahead of time, but computers tend to give very similar results when running repeated benchmark cases, so a low variance should be a safe assumption.

Another tool on Lenth's site is a one-sample t-test. If the variance is assumed to be 0.1, the difference of the means is 0.15 or more, the sample size is 10, and $\alpha = 0.05$, then power = 0.9873. If the difference of the means is much higher, power rapidly approaches 1. The t-test was chosen because the data was expected to be normal and the sample variances would be known. In addition, the variances were expected to be similar. When the results from the testing came in, the variances were indeed similar.

From the results using the two tools on Lenth's site, it would seem that a sample size of 10 should be sufficient. The variances were expected to be low, as computers tend to give similar results when performing the same tasks in repeated runs. During the actual run of the benchmark case tests, variances were very low. In no cases were the variances high enough to lead to an uncertain result when using the t-statistic to compare database products.

For comparison to other research, in Ailamaki, DeWitt, Hill, & Wood (1999), the standard deviation was less than 5%, so the variance may be assumed to be about 0.20. The differences between the means was stated to be significant, so if one assumes that to be 0.2 or more, then it would only take a sample size of 10 to give (using Lenth's one sample t-test tool) a power of 0.8031. In their research however, they opted to keep running benchmarks over and over until the variance was small enough that the differences between the means was significant. They did not state explicitly how many runs they used, but if we assumed they increased the sample size from 10 to 20, the power increases to 0.9886. The researchers did not discuss confidence intervals or power, so it is possible they were not that rigorous in the statistical analysis of their benchmark results.

Methodological Assumptions, Limitations, and Delimitations

The construct validity is good for the environment selected. The benchmark is measuring the performance of a RDBMS in one particular environment, and the measurements reflect only the tasks that make up the benchmark case. Different people may want to adjust the benchmark case, or run an entirely different benchmark. Others may prefer a different operating system or hardware platform.

The internal validity of the research should be strong due to the method used. In particular, the benchmark tests should be able to reproduce the results by simply running through the procedures again. Computer programs can be re-run as many times as desired, so the benchmark testing could be reproduced as needed. Computer performance has almost no variance when measuring benchmark case test runs and starting from the same initial conditions.

Regarding external validity, one has to remember that the performance of a RDBMS under a benchmark may not be indicative of its performance in a given production environment. Different environments have different technical needs, and the particular features of one RDBMS may make it a superior performer over another. For example, a database system at one company may be used mostly for transactional processing, while at another company, a different system may be used mostly for the generation of reports. Users at one company may need to do only simple database queries; users at another company may require complex analytical processing. Another consideration is software; most RDBMS products operate in an established environment with a lot of code written to meet that product's particular syntax. The personnel in place also will greatly affect the performance, as an expert in one RDBMS may not be able to properly tune a RDBMS from a different vendor. An expert database administrator can greatly tune a RDBMS using caching algorithms which can result in vastly improved performance for a particular kind of query, an action that has been used in the past to alter benchmark results (Coffee, 2001). As a result, the results of the benchmark case testing should not be used a sole criterion for which RDBMS is more appropriate for a given setting. It is only one factor amongst many to consider.

The proprietary RDBMS products were originally developed for other operating systems, and then later ported to Linux. As a result, their performance on Linux may not be as optimal as on their original platforms. The database vendors continue to develop and improve the performance of their products on Linux. Linux itself is undergoing some changes to improve its threading performance to make better use of multi-core systems. The benchmark case compares the performance of the RDBMS products on Linux, and individual products may perform better on other platforms.

The benchmark tests were performed on a specific Linux server using a quad-core Intel processor. The performance of the database products on other platforms and configurations may vary. In particular, very large configurations with several processors may give a different result, as open source products generally target more common and affordable configurations.

The size of the test data is another limitation. Database products may perform differently on datasets that are significantly smaller or larger than the sample set used here. Also, on systems where the database will fit completely within the memory cache, one should expect different results, as the disk input/output operations would no longer be a large factor in the performance of the system.

Another limitation regarding benchmark case testing is that most RDBMS products are continuously under development, with new features, functionality, and performance enhancements coming out each year. As a result, any RDBMS product that does not perform well in the present may be well improved in the future. The results of the benchmark case testing are only valid for the versions of the RDBMS products tested; later versions may have different performance and scalability.

Ethical Assurances

The benchmark case tests are performed on neither human nor animal subjects, so there are no concerns as to the ethical treatment of test subjects. No confidential data is used in the benchmark case tests; the test data is randomly generated. The RDBMS software to be used in the benchmark case tests is publicly available. However, certain proprietary RDBMS vendors may not want an unfavorable test score published in association with their trademarks, and hence the proprietary RDBMS vendor names were not identified with the specific experimental results, which were instead be identified by the names *Database A*, *Database B*, and *Database C*.

It is important to mention that every database load is different, and although one database may score higher in the benchmark case tests, it is possible that for a given customer's special needs, another database may be the better performer. Furthermore, database systems support a wide range of tuning options that can affect the system performance. In fact, competitive tuning from the competing RDBMS vendors led to making the TPC-A and TPC-B benchmarks obsolete (Levine et. al., 1993). The purpose of this research is not to provide a new definitive benchmark; the work by the Transaction Processing Performance Council (1992) would be beyond the scope of this research. The goal of this study is to test whether the modern open source RDBMS products are statistically equivalent in performance to their proprietary alternatives, given the experiment's operating system and hardware platform.

Summary

A benchmark case suite of tests was constructed to enable the comparison of the relative performance and scalability of open source and proprietary RDBMS products. This suite enabled the measurement of different aspects of performance, divided into three areas: batch load, transaction processing, and report generation. All of the tests were run on the same hardware and operating system to ensure a fair comparison and reduce the effects of external factors. The database and tests were designed to simulate a real application. Data was collected from each test for every database examined. The results were normalized and then compared to provide an analysis of the relative performance and scalability of the various database products.

CHAPTER 4: FINDINGS

The benchmark case test suite was run on six RDBMS products. For each database system, the batch load, transaction processing, and report generation tests were run 10 times with the system set to one processor core, and then 10 more times using four processor cores. The database systems representing open source RDBMS products were MySQL, PostgreSQL, and Firebird. The proprietary RDBMS products tested were randomly assigned the names Database A, Database B, and Database C. The benchmark values were then derived from this raw data, providing values for the performance of each system using one processor core and four processor cores.

The primary research question, comparing the performance of open source database products to proprietary database products on a system using one processor core, was addressed by averaging the scores for the open source and proprietary database products, respectively. The secondary research question was addressed in a similar fashion, using the performance numbers generated with four processor cores. The third research question compared the scalability of open source database products and proprietary database products by averaging the scalability of the individual products in a similar way to the performance comparison.

After the research questions were addressed, further analysis of the data was performed. This analysis described how the individual database products compared to each other. Both the overall score and the scores of individual tests were examined. Each product's strengths and weaknesses were then described.

Results

The first RDBMS tested was the open source product MySQL. The values in Table 2 are the duration of the each test run in seconds. Lower numbers indicate better performance, because that means the test completed faster. Each test was independent of all other tests, as the database and host computer was restored to the same condition before each test. The results for the four processor tests clearly show some scalability, as the average durations decreased for all three tests.

Table 2

Benchmark Case Test Results for MySQL

	1 Processor			4 Processors		
Run	Load	Transact	Report	Load	Transact	Report
1	7341.067	1000.384	2171.650	6115.579	982.809	1314.101
2	7338.812	986.495	2168.900	6120.227	1044.685	1377.874
3	7317.768	1067.407	2176.624	6125.197	929.285	1335.083
4	7319.580	1074.973	2168.284	6117.780	954.250	1367.676
5	7336.400	1000.923	2170.121	6144.934	950.182	1252.035
6	7329.463	1051.456	2171.027	6136.875	913.952	1293.828
7	7325.987	1258.084	2164.565	6109.646	890.806	1349.081
8	7335.947	1095.969	2161.871	6120.411	946.428	1343.779
9	7328.764	1018.811	2162.720	6124.702	948.495	1348.625
10	7315.200	1050.845	2160.168	6109.059	946.275	1325.243
M	7328.899	1060.535	2167.593	6122.441	950.717	1330.732
SD	9.186	78.250	5.155	11.292	41.253	36.961

In Table 3, the results for PostgreSQL are displayed. Like all other RDBMS products, PostgreSQL displayed some scalability by improving its performance times when the number of processor cores was increased from one to four. PostgreSQL generally outperformed MySQL in all three tests, which was expected given its long history and maturity.

Table 3

Benchmark Case Test Results for PostgreSQL

	1 Processor			4 Processors		
Run	Load	Transact	Report	Load	Transact	Report
1	7142.623	759.968	1754.341	5980.586	698.039	1070.068
2	7135.269	801.655	1720.450	5960.234	752.265	1050.958
3	7139.263	780.662	1706.252	5977.531	895.484	1097.314
4	7142.648	844.259	1765.463	5957.169	720.922	1035.863
5	7142.202	879.241	1650.260	5967.698	739.860	992.061
6	7129.716	821.882	1788.656	5980.632	817.293	1110.021
7	7167.695	776.881	1647.763	5966.340	788.824	1122.986
8	7130.292	847.701	1642.241	5996.537	802.732	1063.312
9	7155.040	822.531	1780.668	5988.692	731.408	1090.703
10	7205.094	869.077	1792.098	5988.484	681.986	1045.314
M	7148.984	820.386	1724.819	5976.390	762.881	1067.860
SD	22.767	40.317	60.483	13.113	64.072	39.150

Firebird did not perform nearly as well as PostgreSQL and MySQL. For example, using one processor core, the batch load test was 48% slower than MySQL and 52% slower than PostgreSQL. The transaction processing test was 4.5% slower than MySQL and 35% slower than PostgreSQL. The report generation test was 48% slower than MySQL and 86% slower than PostgreSQL. This may be the result of weak developer support in the open source community, or a weakness in its overall architecture. Table 4 has the results for the Firebird RDBMS.

Table 4

Benchmark Case Test Results for Firebird

	1 Processor			4 Processors		
Run	Load	Transact	Report	Load	Transact	Report
1	10962.195	1076.200	3318.150	9128.069	920.340	2348.895
2	10763.755	1113.700	3288.241	8212.819	872.904	2319.481
3	10907.235	1007.934	3180.650	9165.509	892.417	2325.369
4	10800.005	1112.657	3143.817	8176.146	914.434	2324.709
5	10916.876	1141.709	3294.253	9177.886	892.430	2328.249
6	10818.974	1076.155	3146.634	8228.598	874.289	2323.482
7	10940.275	1129.909	3167.219	9163.934	889.386	2323.833
8	10864.906	1160.912	3149.617	8223.832	831.808	2311.027
9	10964.951	1118.727	3163.366	9341.931	870.331	2317.843
10	10802.966	1145.591	3298.748	8214.156	834.869	2320.821
M	10874.214	1108.349	3215.069	8703.288	879.321	2324.371
SD	73.915	44.766	74.120	521.993	29.327	9.865

The results for Database A are shown in Table 5. The batch load times for this database were 103% slower than MySQL when using one processor core and 26% slower when using four processor cores. The transaction processing times were 40% faster than MySQL with one processor core and 47% faster with four processor cores. The report generation times were 52% slower than MySQL with one processor core and 120% slower with four processor cores. The batch load times improved dramatically when using four processor cores, completing in almost half the time (7,691 seconds versus 14,843 seconds).

Table 5

Benchmark Case Test Results for the Database A

	1 Processor			4 Processors		
Run	Load	Transact	Report	Load	Transact	Report
1	14582.023	612.840	3558.865	7697.136	452.615	2706.898
2	14467.723	656.124	3182.215	7708.526	507.897	3215.706
3	14734.050	616.912	3099.405	7695.496	502.537	2868.388
4	15193.960	628.670	3378.876	7670.693	506.494	2852.063
5	14879.807	628.720	3093.298	7688.222	489.949	2900.397
6	15185.930	649.394	3496.806	7683.682	506.400	3039.899
7	15105.833	620.545	3632.548	7682.788	468.304	2980.262
8	14598.321	607.427	3020.550	7701.761	541.566	2879.083
9	15236.147	578.759	3290.122	7681.999	510.816	3021.133
10	14455.336	682.499	3303.315	7706.392	523.479	2788.977
M	14843.913	628.189	3305.600	7691.669	501.006	2925.281
SD	315.653	28.821	210.100	12.181	25.580	144.267

Database B outperformed MySQL in both the transaction processing test and the report generation test, but it was a 35% slower in the batch load test when using one processor core and 15% slower when using four processor cores. Database B also showed a significant decrease in durations when going to four processors. The results for Database B are listed in Table 6.

Table 6

Benchmark Case Test Results for the Database B

	1 Processor			4 Processors		
Run	Load	Transact	Report	Load	Transact	Report
1	9899.576	313.202	831.460	7071.768	204.195	376.768
2	9819.985	316.009	434.783	7110.433	205.130	401.915
3	9807.253	329.520	425.166	6995.902	221.915	500.467
4	9767.499	318.472	438.503	7068.226	202.501	396.722
5	10049.453	304.963	488.737	7166.436	227.477	290.623
6	9909.671	315.811	449.354	6993.603	190.545	385.504
7	9744.843	295.662	523.219	6958.228	235.452	427.612
8	9749.717	553.699	421.953	6905.965	192.872	291.846
9	9897.232	771.953	493.121	7189.039	199.217	443.968
10	10021.257	563.602	398.284	7002.834	186.786	369.393
M	9866.649	408.289	490.458	7046.243	206.609	388.482
SD	108.134	163.728	125.745	91.108	16.401	63.965

Finally, the results for Database C are listed in Table 7. Database C had batch load times that were comparable to MySQL, but the transaction processing times were 67% slower when using one processor core and 14% slower when using four processor cores. The report generation times significantly much faster than all other RDBMS products. This may be because the default tuning of this database product is better optimized for query than for transactions. Database C also showed a significant improvement in speed when running on four processor cores compared to one processor core, running the batch load test 17% faster, the transaction processing test 38% faster, and the report generation test 22% faster.

Table 7

Benchmark Case Test Results for the Database C

	1 Processor			4 Processors		
Run	Load	Transact	Report	Load	Transact	Report
1	7230.853	1794.601	345.917	6005.025	1241.378	267.011
2	7230.573	1788.067	345.027	5999.362	936.962	269.695
3	7222.383	1747.011	340.569	5999.994	1336.813	267.999
4	7221.906	1662.639	321.843	5991.588	891.308	262.738
5	7245.330	1755.433	356.955	6010.721	1186.276	267.741
6	7265.503	1739.719	348.941	5999.296	886.703	270.510
7	7217.732	1809.125	329.497	5997.989	956.193	270.717
8	7225.413	1867.167	338.320	5994.920	1230.311	264.561
9	7227.421	1727.128	329.525	6092.024	1090.551	265.177
10	7216.032	1843.963	350.572	5994.043	1091.652	267.206
M	7230.315	1773.485	340.717	6008.496	1084.815	267.336
SD	14.879	59.884	10.991	29.858	161.471	2.610

The benchmark results for one processor are summarized in Table 8. The mean and standard deviation for a given test and RDBMS were normalized into $Y_{navg\text{-}d,1,t}$ by dividing the score by MySQL's score and multiplying by 100. As before, a lower score reflects faster performance, so Database B had the best benchmark score under one processor, and Firebird had the worst.

Table 8

Database Benchmarks for One Processor

	open source			proprietary		
Test	MySQL	Postgre-SQL	Firebird	A	B	C
Load						
$Y_{navg\text{-}d,1,load}$ M	100.000	97.545	148.374	202.539	134.627	98.655
$Y_{navg\text{-}d,1,load}$ SD	0.125	0.311	1.009	4.307	1.475	0.203
Transact						
$Y_{navg\text{-}d,1,trans}$ M	100.000	77.356	104.509	59.233	38.498	167.226
$Y_{navg\text{-}d,1,trans}$ SD	7.378	3.802	4.221	2.718	15.438	5.647
Report						
$Y_{navg\text{-}d,1,report}$ M	100.000	79.573	148.324	152.501	22.627	15.719
$Y_{navg\text{-}d,1,report}$ SD	0.238	2.790	3.419	9.693	5.801	0.507
$Y_{perf\text{-}d,1}$ M	100.000	81.049	124.233	108.709	48.156	111.488
$Y_{perf\text{-}d,1}$ SD	4.059	2.253	2.543	3.333	8.670	3.110

The benchmark for open source RDBMS products when using one processor core is represented by $Y_{open\text{-}perf,1}$. This benchmark is the average of the means $Y_{perf\text{-}d,1}$ for the open source database products. The average of the standard deviations was computed as $[(\sigma_{MySQL}^2+\sigma_{PostgreSQL}^2+\sigma_{Firebird}^2)^{1/2}]/3$. The proprietary RDBMS benchmark score for one processor core was computed in a similar fashion. The t-test was used to compare the two random variables. At an alpha of 5%, the open source RDBMS benchmark $Y_{open\text{-}perf\text{-}1}$ (M=101.761, SD=1.764) was not shown to be significantly lower or equal to the proprietary RDBMS benchmark $Y_{prop\text{-}perf\text{-}1}$ (M=89.451, SD=3.265), t(4)=5.745, p=0.452. Thus, the performance of open source RDBMS products on a system with one processor core was not shown to be better than or equal to the performance of the proprietary RDBMS products.

A similar result occurs when the system is set to run with four processors. In Table 9 the benchmark results are listed. The $Y_{navg\text{-}d,4,t}$ values were normalized to the MySQL one processor values in order to maintain consistency with the rest of the results. This was done by dividing the value by the corresponding MySQL one processor score and multiplying by 100.

Table 9

Database Benchmarks for Four Processors

	open source			proprietary		
Test	MySQL	Postgre-SQL	Firebird	A	B	C
Load						
$Y_{navg\text{-}d,4,load}$ M	83.538	81.546	118.753	104.950	96.143	81.984
$Y_{navg\text{-}d,4,load}$ SD	0.154	0.179	7.122	0.166	1.243	0.407
Transact						
$Y_{navg\text{-}d,4,trans}$ M	89.645	71.934	82.913	47.241	19.482	102.289
$Y_{navg\text{-}d,4,trans}$ SD	3.890	6.042	2.765	2.412	1.546	15.225
Report						
$Y_{navg\text{-}d,4,report}$ M	61.392	49.265	107.233	134.955	17.922	12.333
$Y_{navg\text{-}d,4,report}$ SD	1.705	1.806	0.455	6.656	2.951	0.120
$Y_{perf\text{-}d,4}$ M	80.253	66.575	95.585	82.212	30.513	72.257
$Y_{perf\text{-}d,4}$ SD	2.200	3.367	1.864	2.397	1.242	8.374

The benchmark for open source RDBMS products when using four processor cores is represented by $Y_{open\text{-}perf,4}$. This benchmark is the average of the means $Y_{perf\text{-}d,4}$ for the open source database products. The average of the standard deviations was computed using the same method as for one processor core. The proprietary RDBMS benchmark score for four processor cores was computed in a similar fashion. The t-test was used to compare the two random variables. At an alpha of 5%, the open source RDBMS benchmark $Y_{open\text{-}perf\text{-}4}$ (M=80.804, SD=1.478) was not shown to be significantly lower or equal to the proprietary RDBMS benchmark $Y_{prop\text{-}perf\text{-}4}$ (M=61.660, SD=2.933), t(4)=10.096, p=0.405. Thus, the performance of open source RDBMS products on a system with four processor cores was not shown to be better than or equal to the performance of the proprietary RDBMS products.

The scalability benchmark results are listed in Table 10. Because the shorter duration using four processor cores is divided by the longer duration using one processor core, lower values for scalability represent more speedup. All of the proprietary database systems had scalability scores that were better than the scalability scores of the open source database systems. The standard deviation of the individual scalability scores was computed by finding the variance of a ratio of two random variables (Kalton, 1983). The benchmark for open source scalability is represented by $Y_{open\text{-}scale}$ and is computed as the average of the scalability scores for the open source database products. The average of the standard deviations was computed using the same method as for the performance benchmarks. The scalability benchmark for proprietary database systems is represented by $Y_{prop\text{-}scale}$ and its mean and standard deviation were computed in a similar fashion. The t-test was used to compare the two random variables. At an alpha of 5%, the open source

RDBMS benchmark $Y_{open\text{-}scale}$ (M=0.798, SD=0.022) was not shown to be significantly lower or equal to the proprietary RDBMS benchmark $Y_{prop\text{-}scale}$ (M=0.679, SD=0.048), t(4)=3.904, p=0.347. Thus, the scalability of open source RDBMS products was not shown to be better than the performance of the proprietary RDBMS products.

Table 10

Scalability of the Database Systems

	open source			proprietary		
	MySQL	Postgre- SQL	Firebird	A	B	C
$Y_{scale\text{-}d}$ M	0.803	0.821	0.769	0.756	0.634	0.648
$Y_{scale\text{-}d}$ SD	0.039	0.047	0.022	0.032	0.117	0.077

In all three cases, the results favored the proprietary benchmarks. From the viewpoint of the benchmarks, proprietary products, in general, were better at performance and scalability than their open source competition.

Evaluation of Findings

Because proprietary database systems outperform open source database systems, technology managers will be justified in selecting them when choosing a database product, if all other factors are equal. In most cases, however, performance is not the only concern; technology managers also have to be cognizant of the costs of the products and their related services. The performance benchmark numbers were close, with the proprietary systems running 12% faster under one processor core and 24% faster under four processor cores. There are many applications where having a small increase in speed would not be worth the much higher costs of going with proprietary systems.

Because of the superior scalability of the proprietary database systems, the performance difference between open source and proprietary database products increases with the number of processor cores. Technology managers implementing projects on larger systems with more processor cores will see proprietary database systems have more of a performance advantage over open source database systems compared to smaller applications that run on a small server with one processor core. The more processors the server has, the greater the difference in performance between proprietary and open source solutions, which is a strong argument in favor of proprietary RDBMS products.

Open source database products do not have license fees, and thus the cost to maintain them does not increase with the number of processor cores. Proprietary database systems are licensed per processor core, and thus they have a linear increase in cost with a less than linear increase in performance. Technology managers have to analyze their own situation and determine whether the increase in performance is worth the increase in cost.

When the performance data of individual database products was examined, the general statement that proprietary database systems outperform open source database systems no longer held true. As can be seen in Figure 2, MySQL and PostgreSQL were faster than Database A and Database C when using one processor core. In this figure and the following figures, lower bars reflect shorter duration and better performance, with times normalized to a percentage of MySQL's performance with one processor core. When using four processor cores, PostgreSQL was faster than Database A and had overlapping confidence bars with Database C. While Database B clearly had the best performance, it might not be the incumbent product in a given organization. For example, a manager who has licenses available for Database A may only want to compare Database A to MySQL and PostgreSQL, both of which would give him better performance. Alternately, a different manager who has licenses for Database B available would not see any performance improvement by switching to a different database product.

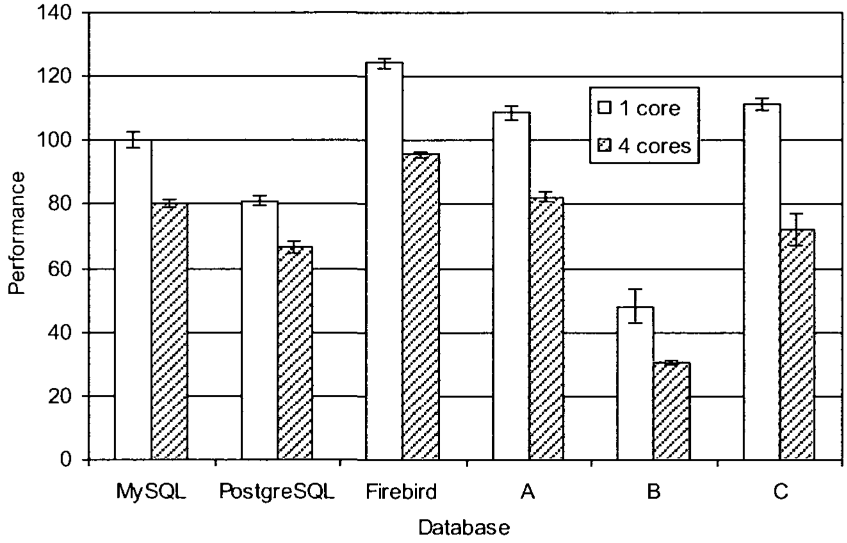

Figure 2. Performance of database products when using one and four processor cores.

After examining the scalability of individual database products, the proprietary database systems were shown to have more of an advantage over the open source database systems, as can be seen in Figure 3. The only exception was Firebird, which has similar scalability to Database A. Database A still had the advantage, because Firebird started out with slower performance than Database A, and since the scalability of the two systems is similar, it was unlikely that the performance of Firebird would ever catch up to Database A. As a result, when scalability is a consideration for technology managers, the proprietary database products were shown to provide more speedup.

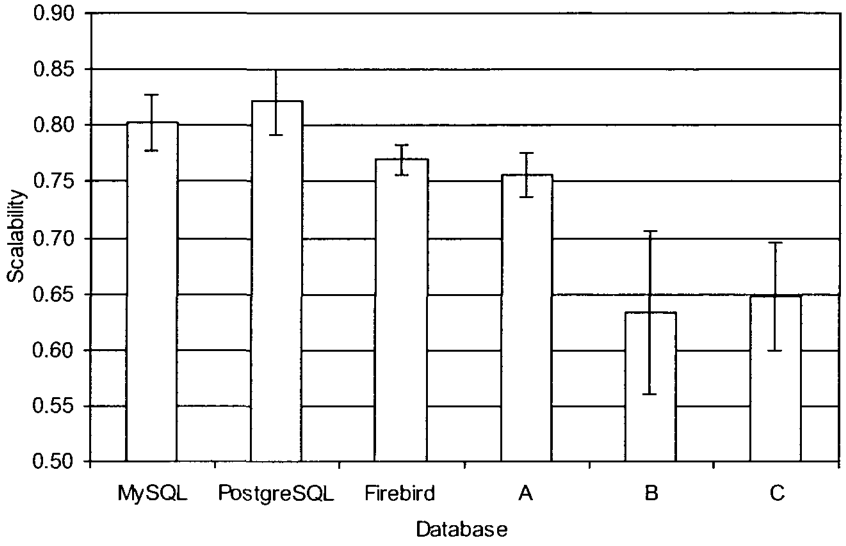

Figure 3. Scalability of database products.

The performance benchmark test suite consisted of three tests which measured different types of database activity, which were the batch load test, the transaction processing test, and the report generation test. The six database products tested had different strengths and weaknesses in the three different tests. Because no single product had the best performance in all three tests, it would be beneficial for a technology manager to adjust the weights of the tests. In this research, the weights chosen were 15% for batch load, 55% for transaction processing, and 30% for report generation. A manager of an online analytical processing application, for example, would be more concerned with the query tests. A manager of such an application might choose weights of 10% for batch load, 5% for transaction processing, and 85% for report generation, which would lead to much different results in the benchmark. As there are an effectively unlimited number of different combinations for weights, it was helpful to analyze the individual test results separately.

The batch load performance test results are shown in Figure 4. This test was focused on insertion activity, similar to what occurs in a batch load from one system to another, or in a recording system. MySQL, PostgreSQL, and Database C had the best scores here. Database A was the slowest when running with one processor core, but its performance improved dramatically with four processor cores. For this test, the open source products were highly competitive, with MySQL and PostgreSQL performing as well as or better than all of the other database products.

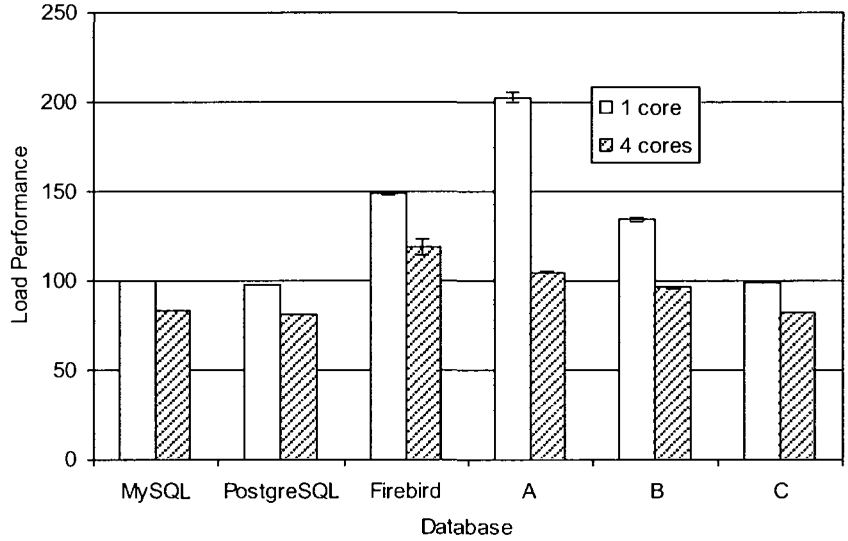

Figure 4. Batch load performance of database products.

The transaction processing test results are shown in Figure 5. This test consisted of several different types of transactions that would insert, update, or delete data, often while locking one or more tables. The RDBMS with the best lock manager and a superior architecture for handling concurrent processing would complete the test in the shortest amount of time and outperform the others. Here, both Database A and Database B outperformed all of the open source database products. MySQL, for example, was more than twice as slow as Database B when using one processor core, and more than four times slower when using four processor cores. Database C, however, was slower than all of the open source database products, and should be avoided if possible for those applications that rely heavily on transaction activity.

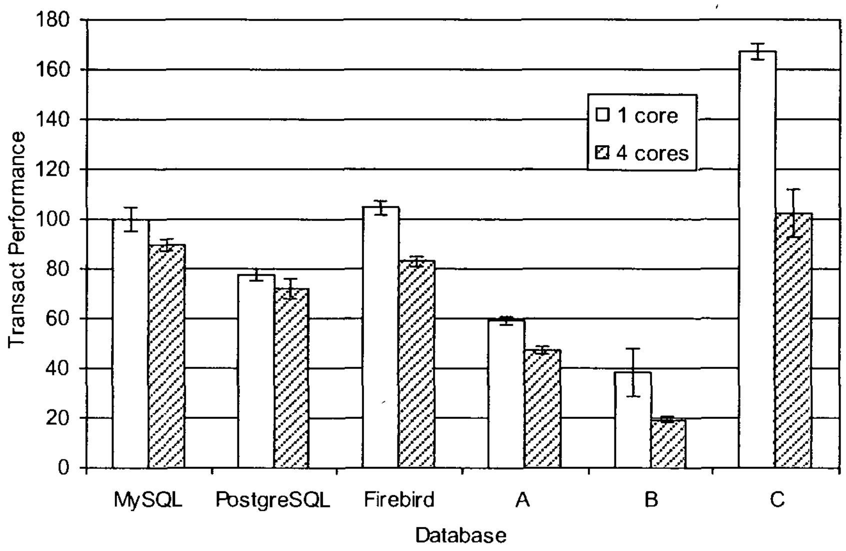

Figure 5. Transaction processing performance of database products.

The report generation test results are shown in Figure 6. Here, Database C outperformed all other systems. Because Database C performed poorly in the transaction processing test and yet outperformed the competition in the report generation test further emphasizes the need for a technology manager to select weights that are appropriate to the application. For report generation, Database A was the slowest database system, taking more time to complete than all of the open source products. The difference in speed between the open source products and the proprietary products Database B and Database C is significant, with the open source solutions being several times slower.

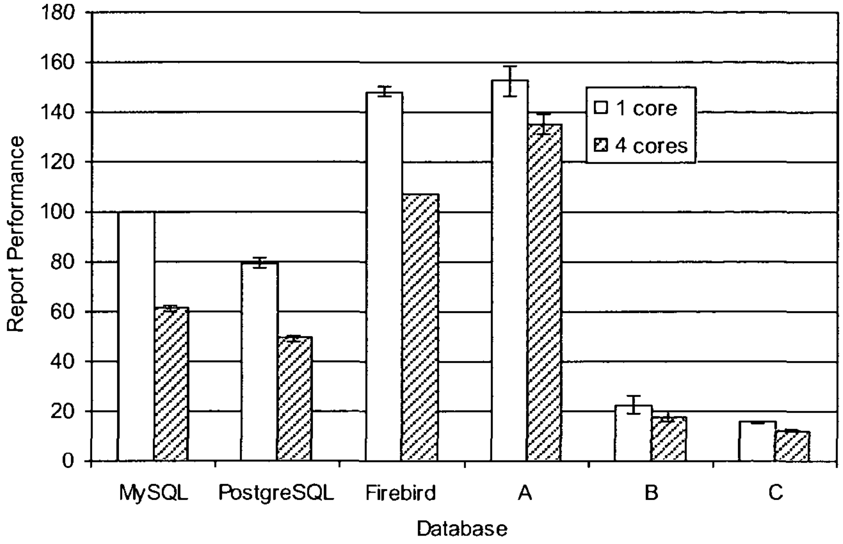

Figure 6. Report generation performance of database products.

All of the database systems had their strengths and weaknesses. Some were stronger with a particular test, or had better scalability. All of the RDBMS products were highly configurable and could be improved with complex performance tuning methods. A highly experienced database administrator with skills in this area could greatly improve the performance for a given application. As a result, small differences in benchmark numbers may be less important than the overall skill of the database administrator. Technology managers should consider the level of skill of the database administrators available to them when considering making a choice between RDBMS products.

Summary

In general, the proprietary database systems tested had better performance and scalability than the open source database systems. This general statement did not apply to specific database products, as individual scores vary. Different products were stronger based on the kind of application being tested. This research measured the performance of RDBMS products under batch load, transaction processing, and report generation. The relative strengths and weaknesses of the database products varied with each test. Technology managers should weight the results of the performance benchmarks according to the needs of the application being implemented.

CHAPTER 5: IMPLICATIONS, RECOMMENDATIONS, AND CONCLUSIONS

Technology managers need a way to measure the relative performance and scalability of the various proprietary and open source database products available to them. A suite of benchmark case tests was created to provide a tool for the use in measuring performance and scalability. This suite was run against three proprietary database products and three open source database products. The tests were all run on the same hardware and operating system, to provide a fair comparison between the products. The test platform was a Linux server, which prevented the testing of Microsoft SQL Server. The results of the tests run in this research will quickly become outdated as new versions of database software become available. New database products will also be brought to market over time. Technology managers may use the techniques described in this paper to do their own testing. What follows are the implications and recommendations based on the findings of this research.

Implications

The t-test was used to compare the group of open source RDBMS products against the group of proprietary RDBMS products. The t-test was selected because the data sampled from multiple runs of the benchmarks was expected to be normal with low variances, because computers generally perform the same task in the same amount of time. This proved to be true in the results, and thus having only 10 runs per test was satisfactory for the purpose of comparing the benchmark scores.

Proprietary database systems were 12% faster, on average, than open source database systems when using one processor core. This answers the first research question, on a server with one processor core, to what extent, if any, does the performance of open source RDBMS products, on average, equal or exceed the performance of proprietary RDBMS products, on average, when run on the same operating system and hardware? The answer is that open source RDBMS products do not equal or exceed the performance of proprietary RDBMS products when running on a single-core system. The difference in performance may not justify the higher costs of proprietary products, but that decision depends on each individual technology manager's situation. For some technology managers, a 12% difference in performance may not be enough to justify the higher costs of using proprietary software.

When using four processor cores, the proprietary database systems were 24% faster, on average, than the open source database systems. This answers the second research question, on a server with four processor cores, to what extent, if any, does the performance of open source RDBMS products, on average, equal or exceed the performance of proprietary RDBMS products, on average, when run on the same operating system and hardware? The answer is that open source RDBMS products do not equal or exceed the performance of proprietary RDBMS products when running on a system with four processor cores. The difference in performance was greater than with one processor core, so the benefit of using proprietary RDBMS products appears to increase as more cores are added to a system. Technology managers concerned about performance on quad-core systems would get better results with proprietary RDBMS products.

The scalability of proprietary database systems was 15% better than the scalability of open source database systems. This answers the third research question, to what extent, if any, does the scalability, from one processor to four processor cores, of open source RDBMS products, on average, equal or exceed the scalability of proprietary RDBMS products, on average, when run on the same operating system and hardware? The answer is that the scalability of open source RDBMS products does not equal or exceed that of proprietary RDBMS products. Because the scalability of proprietary products is superior, one would expect the difference in performance between proprietary and open source RDBMS products to increase as more processor cores are added. Technology managers looking to create large database systems with many processor cores will get better performance from the proprietary products.

The proprietary database systems were shown to be superior for all three research questions. For technology managers not concerned with licensing costs, the faster proprietary products are a superior solution. Cost-sensitive technology managers have to consider the benefits of the small increase in speed against the much higher costs of the proprietary database licenses. For some managers, the difference in performance may not justify the cost of proprietary RDBMS licenses.

Open source RDBMS products were closest in performance to the proprietary RDBMS products when operating on a system with one processor core. This makes them a better candidate for consideration in an environment with an array of lightweight database servers. Because proprietary RDBMS licenses are paid per processor core, implementing a large grid of small database servers can be very expensive when using proprietary RDBMS software. Some proprietary database solutions become very expensive as newer four-core and eight-core processors become available (Pallatto, 2005).

The benchmark case test suite measured three aspects of database performance: batch load, transaction processing, and report generation. Each database product had its strengths and weaknesses, as reflected in the individual test scores. The individual scores are more relevant than the general scores when a technology manager needs to compare the performance of two specific database products. Because each product scored differently under each test, the weights used in the benchmark greatly affect the outcome. If different weights are used, the results could favor different database products.

The results of the individual RDBMS benchmark scores indicate that the poor performance of the Firebird RDBMS impacted the benchmark results for open source RDBMS products. As can be seen in Figure 2, MySQL and PostgreSQL, when operating with one processor, actually had better performance than Database A and Database C. With four processors, the performance of MySQL and PostgreSQL was similar to that of Database A and Database C. Database B, however, had the best performance of all of the products.

Unfortunately for the open source RDBMS products, none of them compared favorably to the proprietary RDBMS on an individual basis. The only one that came close was Firebird, which had similar scalability to Database A. Firebird starts out with poor performance however, so being able to scale does not necessarily help it when comparing against other RDBMS products.

The various RDBMS products performed differently for each benchmark case test. In the batch load performance test, MySQL and PostgreSQL outperformed Database A and Database B, and had similar performance to Database C. So for an application that is mostly oriented around batch load, MySQL and PostgreSQL would actually be superior solutions. One examples of a batch load application is an action logging system, where events are recorded for auditing purposes but infrequently queried.

The transaction processing test had interesting results for open source RDBMS products. The performance of all three open source database products in the transaction processing test was better than Database C. Database A and Database B, however, had much better performance than all of the open source products. This is a good example of how individual database scores give much different results than the general scores for the group.

In the report generation benchmark case test, Database B and Database C performed much better than the other four database products. MySQL and PostgreSQL performed better than Database A. Although proprietary database systems outperform open source database systems as a group, the performance of individual database products varies.

Recommendations

RDBMS license fees can be very expensive, as these products are licensed per core. Technology managers looking to implement a large network of database systems could face very large capital expenditures when purchasing licenses to proprietary database products. For some applications, the performance increase achieved by using proprietary database products could justify the cost. Other technology managers may not be sensitive to price for other seasons, due to site licenses or an excess inventory of licenses. In these cases, technology managers would be wise to select proprietary database products and take advantage of their superior speed and scalability.

Technology managers who have applications with specific types of loads may find it advantageous to select MySQL or PostgreSQL in certain cases. The results of the batch load benchmark case test indicated that MySQL and PostgreSQL would give better performance than Database A and Database B, and similar performance to Database C. A technology manager with an application that does a lot of batch load processing should consider MySQL and PostgreSQL.

The transaction processing benchmark case test results indicated all three open source database products to be superior to Database C. Database A and Database B gave better performance than all of the open source products. A technology manager looking for transaction processing performance should avoid Database C, regardless of the cost. The open source products would only be a good fit if the costs of Database A and Database B was too expensive to consider.

The difference in performance for report generation was significant. It would be hard to justify open source solutions for applications doing this kind of work unless performance was not a major factor. If a technology manager already had Database A installed, however, then it would be possible to increase performance by adopting a low-cost open source solution, because only Database B and Database C gave superior performance.

If a technology manager had the time and resources to run the benchmark case tests on the technology manager's own systems, then more relevant results could be obtained. By running this benchmark on different hardware with different numbers of processing cores and a different version of Linux, different and more relevant results may be obtained. Furthermore, the benchmark could compare specific, named versions of proprietary RDBMS products instead of the blind names for Database A, Database B, and Database C.

For those managers who are under pressure to reduce technology costs, selecting an open source database product could save tens of thousands of dollars per system. Although the performance of these products is less than the proprietary database products, the difference may not be significant enough to justify the cost. The technology manager would also be free to add additional processor cores and to install the open source RDBMS products on other servers without incurring additional fees. This gives the technology manager the freedom and flexibility to respond to changes in demand.

The performance of the database products can also be improved through fine-tuning by expert database administrators. One difference between these database products was the difficulty of installation. MySQL was the easiest to install and configure, while the proprietary systems proved to be more challenging. The challenges with installing and fine-tuning RDBMS products should be considered when comparing products. This qualitative difference is difficult to measure, and may not be a factor at all if a technology manager has a database specialist available to do this kind of work. In a smaller organization such specialists may be rare and hard to obtain.

The benchmark tests performed in this research were run on one specific server, using versions of the RDBMS software that were current at the time the measurements were taken. Any given technology manager is likely to work with different hardware and operating systems. Newer versions of open source and proprietary database products come out every year, possibly providing improved performance and scalability. Individual database systems can also be tuned for better performance if the services of an expert database administrator are available. These factors could all lead to different benchmark results.

To provide a more relevant measurement, the benchmark case tests can be run on a technology manager's own hardware and operating systems. The test weights in the benchmark can be adjusted to reflect the activity of a specific application. The newest versions of each database product can be used in each manager's own tests, and the local database administrator can improve the performance by tuning the database. Following these steps will result in benchmark measurements more relevant to a specific environment.

As systems with more processor cores become available, future research may be done by re-running this benchmark on database products, measuring the performance improvement as processor cores are added. It is possible that the scalability of database products decreases as more processor cores are added to a system, due to increased contention for shared resources. The scalability of database products may also be affected by the underlying operating system, which can be changed in future testing. If the tests are run on a Microsoft Windows server, then Microsoft SQL Server could be compared as well.

Another area for future research is an examination of the performance and scalability of other products, including object-oriented database products, embedded database systems, and hierarchical database systems. Newer versions of PostgreSQL from EnterpriseDB may provide different results than the original source. MySQL, which can use multiple data storage engines, can be tested separately with each engine. New database products will be introduced to the market over time, giving technology managers more options to consider. As the options increase, the need for relevant benchmarks becomes even greater.

Conclusions

Technology managers will find superior performance using the proprietary database products. As a general statement, the performance and scalability of open source products does not surpass the proprietary database products. If technology managers examine individual products more closely, they will find some open source products compare well for certain kinds of tests. The costs of the proprietary products may be a significant factor for some technology managers, who may find the difference in performance not worth the higher cost of the proprietary database products. By using the techniques described in this paper, the tests can be customized to more closely reflect an individual technology manger's application load.

REFERENCES

Aberdour, M. (2007). Achieving quality in open source software. *IEEE Software, 24*(1), 58-64. Retrieved November 23, 2007, from ABI/INFORM Global database. (Document ID: 1326207451).

Aczel, A. D. & Sounderpandian, J. (2002). *Complete business statistics* (5th ed.). (pp. 282-371). Chicago, IL: Irwin.

Alfs, G. (2007). News flash: Intel details upcoming new processor generations. Retrieved July 7, 2007, from http://www.intel.com/pressroom/archive/releases/Intel_New_Processor_Generations.pdf

Ailamaki, A. G., DeWitt, D. J., Hill, M. D., & Wood, D. A. (1999, September). DBMSs on modern processors: Where does time go? *The VLDB Journal*, 266-277.

AMD (2007, November 30). AMD demonstrates world's first native quad-core X86 server processor. Retrieved July 7, 2007, from http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_13744~114496,00.html

Apicella, M., Biggs, M. (2000, April 17). Uncovering database benchmarks. *InfoWorld, 22,* 63-64.

Biggs, M. (2002, September 23). Finding an opening. *InfoWorld, 24*, 19-20.

Boulton, C. (2003, June 16). Are open source databases following Linux's footsteps? *CIO Update*. Retrieved August 18, 2006, from http://www.cioupdate.com/trends/article.php/2222231

Campbell, S. (2002). IBM solidifies Linux strategy. CRN, 1009, 12.

Caniano, S. (1988). All TP1s are not created equal. *Datamation, 34*, 16 (51-54).

Chabrow, E. (2008, January 15). The new IT worker shortage. *CIO Insight.* Retrieved January 21, 2008 from http://www.cioinsight.com/article2/0,1540,2248193,00.asp

Chen, A. (2002, July 8). Open source gets IT scrutiny. *eWeek*, 19.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM, 13,* 377-387.

Coffee, P. (2001, November 19). Handicapping the hardware. *eWeek.* Retrieved June 5, 2006, from http://www.eweek.com/article2/0,1759,1010924,00.asp

Coffee, P. (2005, September 26). Reinventing enterprise technology. *eWeek*. Retrieved July 7, 2007, from http://www.eweek.com/article2/0,1759,1862988,00.asp

Cohen, N. (2003, August 27). Blades, clusters, and big paybacks. *Open: The strategic guide to open source*. Retrieved September 1, 2003, from http://www.open-mag.com/936339824.shtml

Cornetto, J. (1998, September). IBM, Sybase latest to put RDBMS on Linux. *InfoWorld, 20*(39), 14. Retrieved December 22, 2007, from ABI/INFORM Global database. (Document ID: 34590667).

Cox, J. (2004, March). Open source database improvements grow. *Network World, 21*(11), 32. Retrieved December 6, 2007, from ABI/INFORM Global database. (Document ID: 580863321).

D'Agostino, D. (2005, November 23). Open source turns strategic. *CIO Insight, 1*(60), 65-77.

DeWitt, D. J. (1993). The Wisconsin benchmark: Past, present, and future. In J. Gray (Ed.), *The benchmark handbook for database and transaction processing systems, 2nd ed.* (pp. 269-308). San Mateo, CA: Morgan Kaufman Publishers.

Dickerson, C. (2003, May 19). If the glass slipper fits... *InfoWorld, 25*, 32.

Donston, D. (2002, July 8). Open-source enterprise. *eWeek*, 19.

Dyck, T. (2002, March 26). Clash of the titans: SQL databases. *PC Magazine*. Retrieved June 5, 2006, from http://www.pcmag.com/article2/0,4149,4178,00.asp

Dyck, T. (2003, July 7). Database server clash revisitied. *eWeek*. Retrieved June 5, 2006, from http://www.eweek.com/article2/0,1759,1184846,00.asp

Ebert, C. (2007). Open source drives innovation. *IEEE Software, 24*(3), 105. Retrieved November 23, 2007, from ABI/INFORM Global database. (Document ID: 1326654061).

Elmasri, R. & Navathe, S. (1994). Fundamentals of database systems (2nd ed.). (pp. 259-260). Redwood City, CA: The Benjamin/Cummings Publishing Company.

Florescu, D., & Kossman, D. (2009). Rethinking cost and performance of database systems. *SIGMOD Record, 38,* 1.

Gillespie, M. (2007, May 8). Transitioning software to future generations of multi-core. Retrieved July 7, 2007, from http://softwarecommunity.intel.com/articles/eng/1273.htm

Gray, J. (1993). Introduction. In J. Gray (Ed.), *The benchmark handbook for database and transaction processing systems, 2nd ed.* (pp. 1-19). San Mateo, CA: Morgan Kaufman Publishers.

Gray, J. & Nyberg, C. (1994). Desktop batch processing. *Proceedings of the IEEE Spring CompCon, 1994.*

Groff, J. (2002). SQL: The complete reference (2nd ed.) (pp. 29-31, 920-921). Blacklick, OH: McGraw-Hill Professional.

Hicks, M. (2002, August 19). Open-source databases hike enterprise appeal: Developers improve support for transactions, recovery, replication. e*Week,* 12.

Kalton, G. (1983). Introduction to survey sampling (p. 44). Thousand Oaks, CA: Sage Publications.

Krill, P. (2002, September 23). IBM, MS reject MySQL: Open-source database growth draws fire from industry stalwarts. *InfoWorld.* Retrieved August 3, 2003, from http://www.findarticles.com

LaMonica, M. (2005). Database vendors eye open-source effect. *CNET News.com.* Retrieved July 31, 2005 from http://news.com.com/2100-1012_3-5785645.html

Lenth, R. V. (2006). Java applets for power and sample size [Computer software]. Retrieved January 24, 2007, from http://www.stat.uiowa.edu/~rlenth/Power

Levine, C., Gray, J., Kiss, S., & Kohler, W. (1993). The evolution of TPC-benchmarks: Why TPC-A and TPC-B are obsolete. *SFSC Technical Report 93.1,* Digital Equipment Corporation. Retrieved June 3, 2006, from http://research.microsoft.com/~gray/papers/TPC_Evolution.doc

Loney, K. & Kotch, G. (2000). Oracle8i: The complete reference (10th ed.). Berkeley, CA: Osborne/McGraw-Hill.

Martens, C. (2007, July 27). SourceForge unveils the winners of the "open-source Oscars". *NetworkWorld.* Retrieved December 6, 2007 from http://www.networkworld.com/news/2007/072707-sourceforge-unveils-the-winners-of.html

Matoria, R. K. & Upadhayay, P. K. (2002). Design and development of web-enabled databases in libraries with special reference to RDBMS: selection of tools and technologies. *DESIDOC Bulletin of Information Technology, 22*(4), 9-15. Retrieved July 23, 2003, from WilsonSelectPlus.

Mears, J. (2005). Open source databases grow. *Network World, 22*, 34, 21-22.

Mitchell, R. (2006, August 28). Solid-state disk: Soul of the new machines. *Computerworld,* 40.

Niccolai, J. (2006, February 7). MySQL buys company, hires noted database architect. *NetworkWorld.* Retrieved December 6, 2007 from http://www.networkworld.com/news/2006/022706-mysql-buys-netfrastructure.html

Niccolai, J. (2007, April 23). EnterpriseDB upgrade aimed at Oracle. *NetworkWorld.* Retrieved December 6, 2007 from http://www.networkworld.com/news/2007/042307-enterprisedb-upgrade-aimed-at.html

Oppel, A. J. (2004). Databases demystified (pp. 17-19). Blacklick, OH: McGraw-Hill.

Pallatto, J. (2005, July 21). Oracle multicore licensing ignores market reality. *eWeek.* Retrieved August 20, 2005, from http://www.eweek.com/article2/0,1759,1839671,00.asp

Poess, M. and Floyd, C. (2000) New TPC benchmarks for decision support and web commerce. *SIGMOD Record, 29.* Retrieved July 27, 2003, from http://www.acm.org/sigmod/record/issues/0012/standards.pdf

Raymond, E. (2001). The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary. Cambridge, MA: O'Reilly.

Register Research (2003, November 7). IBM and HP take phony benchmark war up several notches. The Register. Retrieved August 18, 2006, from http://www.theregister.co.uk/2003/11/07/ibm_and_hp_take_phony/

Scannell, E., Sullivan, T. (2000, October 23). New front opens in database war. *InfoWorld, 22,* 12.

Serlin, 0. (1993). The history of the DebitCredit and the TPC. In J. Gray (Ed.), *The benchmark handbook for database and transaction processing systems, 2nd ed.* (pp. 21-40). San Mateo, CA: Morgan Kaufman Publishers.

Silberschatz, A., Korth, H. F., Sudarshan, S. (2002). Database system concepts (4th ed.) New York, NY: McGraw-Hill.

Silwa, C. (2005). Financial services companies lead the charge to Linux. *Computerworld, 39,* 29, 34-35.

Songini, M. L. (2003, May 5). Oracle database users' interest in Linux grows. *Computerworld, 37,* 6.

Songini, M. L. (2004, July 12). MySQL, SAP plot upgrade of open-source database. *Computerworld, 38*(28), 10.

Strandell, T. (2003, May 8). *Open source database systems: Systems study, performance, and scability.* Unpublished master's thesis. University of Helsinki, Helsinki, Finland. Retrieved July 31, 2005, from http://www.cs.helsinki.fi/u/tpstrand/thesis/Toni_Strandell_Masters_Thesis.pdf

Spooner, J. G. (2005, July 1). Intel's dual-core servers prepare for their close-up. *eWeek.* Retrieved August 20, 2005, from http://www.eweek.com/article2/0,1759,1833924,00.asp

Transaction Processing Performance Council (TPC) (1992). TPC-benchmark A: Standard specification revision 1.1. In J. Gray (Ed.), *The benchmark handbook for database and transaction processing systems, 2nd ed.* (pp. 41-85). San Mateo, CA: Morgan Kaufman Publishers.

Witten, B., Landwehr, C., & Caloyannides, M. (2001). Does open source improve system security? *IEEE Software, 18*(5), 57-61. Retrieved November 25, 2007, from ABI/INFORM Global database. (Document ID: 81568046).

Yarger, R. J., Reese, G., & King, T. (1999). MySQL & mSQL. Sebastopol, CA: O'Reilly.

APPENDICES

Appendix A:

Generation Script for States

```
#!/usr/bin/perl -w
#
# states.pl
#

@states = ("AK","AL","AR","AZ","CA","CO","CT","DE","FL","GA","HI","IA",

"ID","IL","IN","KS","KY","LA","MA","MD","ME","MI","MN","MO","MS",

"MT","NC","ND","NE","NH","NJ","NM","NV","NY","OH","OK","OR","PA",
        "RI","SC","SD","TN","TX","UT","VA","VT","WA","WI","WV","WY");

open(STATES,">../raw/states.csv");
for ($i=0;$i<50;$i++) {
        $foo = int(rand()*20) + 52;
        printf STATES ("\"%s\",%.3f\n", $states[$i], $foo/8);
}
```

Appendix B:

Generation Script for Departments

```
#!/usr/bin/perl -w
#
# departments.pl
#

@a = ("Home", "Office", "Outdoor", "Travel", "Discount", "All-Weather",
        "Sporting", "Premium", "Medical", "Professional", "Home-Made");
@b = ("Furniture", "Clothing", "Soft Goods", "Men's Wear", "Women's
Wear",
        "Goods", "Books", "Appliances");

$remaining = 60;
$key = 1;
open(DEPTS,">../raw/departments.csv");
while($remaining) {

        $a_int = int(rand()*11);
        $b_int = int(rand()*8);

        $name = "$a[$a_int] $b[$b_int]";

        if(defined($taken{$name})) {
                next;
        } else {
                print DEPTS "$key,\"$name\"\n";
                $taken{$name} = 1;
                $remaining--;
                $key++;
        }
}
```

Appendix C:

Generation Script for Stores

```
#!/usr/bin/perl -w
#
# stores.pl
#

@a = ("Mega", "Super", "Mini", "Shop");
@b = ("Mart", "Online", "Catalog");

@states = ("AK","AL","AR","AZ","CA","CO","CT","DE","FL","GA","HI","IA",

"ID","IL","IN","KS","KY","LA","MA","MD","ME","MI","MN","MO","MS",

"MT","NC","ND","NE","NH","NJ","NM","NV","NY","OH","OK","OR","PA",
        "RI","SC","SD","TN","TX","UT","VA","VT","WA","WI","WV","WY");

$remaining = 250;
$key = 1;
open(STORES, ">../raw/stores.csv");
while($remaining) {

        $a_int = int(rand()*4);
        $b_int = int(rand()*3);

        $st_int = int(rand()*50);

        $d = rand();
        if($d < 0.8) {
                $discount = "0.00";
        } elsif($d < 0.85) {
                $discount = "0.05";
        } elsif($d < 0.90) {
                $discount = "0.10";
        } elsif($d < 0.95) {
                $discount = "0.15";
        } else {
                $discount = "0.20";
        }

        $name = "$a[$a_int] $b[$b_int]";
        $taken{$name}++;

        printf STORES ("$key,\"$name\ #%d\",$discount,%s\n",
                $taken{$name}, $states[$st_int]);
        $remaining--;
        $key++;
}
```

Appendix D:

Generation Script for Department Discounts

```
#!/usr/bin/perl -w
#
# department_discounts.pl
#

$n_stores = 250;
$n_departments = 60;

open(DDISC, ">../raw/department_discounts.csv");
for($i=1; $i<=$n_stores; $i++) {
  for($j=1; $j<=$n_departments; $j++) {
    if(rand() < 0.2) {

        $d = rand();
        if($d < 0.5) {
                $discount = "0.00";
        } elsif($d < 0.60) {
                $discount = "0.05";
        } elsif($d < 0.70) {
                $discount = "0.10";
        } elsif($d < 0.80) {
                $discount = "0.20";
        } elsif($d < 0.90) {
                $discount = "0.30";
        } else {
                $discount = "0.40";
        }

        print DDISC "$i,$j,$discount\n";

    }
  }
}
```

Appendix E:

Generation Script for Shipping

```
#!/usr/bin/perl -w
#
# shipping.pl
#

@states = ("AK","AL","AR","AZ","CA","CO","CT","DE","FL","GA","HI","IA",
         "ID","IL","IN","KS","KY","LA","MA","MD","ME","MI","MN","MO","MS",
         "MT","NC","ND","NE","NH","NJ","NM","NV","NY","OH","OK","OR","PA",
         "RI","SC","SD","TN","TX","UT","VA","VT","WA","WI","WV","WY");
@weights = (5,10,20,50,100,150,200,250,300,400);
@costs = (6.0,8.0,15.0,30.0,55.0,75.0,90.0,115.0,130.0,150.0);

open(SHIPPING, ">../raw/shipping.csv");
for ($i=0;$i<50;$i++) {
  for ($j=0;$j<50;$j++) {
    $factor = 1.0 + (int(rand()*6) + 1) * 0.1;
    for ($k=0;$k<10;$k++) {
        $adjusted_cost = $factor * $costs[$k];
        print SHIPPING "\"$states[$i]\",\"$states[$j]\","
        . "$weights[$k],$adjusted_cost\n";
    }
  }
}
```

Appendix F:

Generation Script for Items

```
#!/usr/bin/perl -w
#
# items.pl
#

@adjective = ("Fast", "Slow", "Premium", "Cool", "Hot", "Stylish",
        "Sophisticated", "Gentle", "Rough");
@noun = ("Sweater", "Coat", "Pants", "Shoes", "Socks", "Jacket",
"Suit",
        "Sofa", "Chair");
@color = ("Red", "Blue", "Yellow", "Green", "Purple", "Orange", "Pink",
        "Black", "White", "Gray");
@size = ("Small", "Medium", "Large", "Petite", "Big", "Tall", "S",
"XS",
        "M", "L", "XL", "Junior");
@brand = ("Smyth", "Jonez", "Lightyear", "Ralf", "GKline", "Gooch");
@style = ("Modern", "Spring", "Fall", "Summer", "Winter", "Casual",
        "Business");
@cloth = ("Felt", "Cotton", "Silk", "Nylon", "Rayon", "Leather",
"Corduroy",
        "Chenille", "Elastic", "Spandex");

$words[0] = \@adjective;        $n_words[0] = $#adjective;
$words[1] = \@noun;     $n_words[1] = $#noun;
$words[2] = \@color;    $n_words[2] = $#color;
$words[3] = \@size;     $n_words[3] = $#size;
$words[4] = \@brand;    $n_words[4] = $#brand;
$words[5] = \@style;    $n_words[5] = $#style;
$words[6] = \@cloth;    $n_words[6] = $#cloth;

open(ITEMS, ">../raw/items.csv);
for($i=1; $i<=25000; $i++) {

        $words = int(rand()*4) + 2;
        $name = "";
        for ($w=0;$w<$words;$w++) {
                $wordtype = int(rand()*7);
                $word_int = int(rand()*$n_words[$wordtype]);
                $name .= "$words[$wordtype][$word_int] ";
        }
        chop($name);
        if(length($name) > 40) {
                $name = substr($name, 0, 40);
        }

        $price = 10.0 + (rand()*6  + 1) * (rand()*100);
        $weight = 1 + int(rand()*20);
        $d = rand();
        if($d < 0.2) {
                $discount = 0.05 * (int($d * 5 * 4) + 1);
        } else {
                $discount = 0.00;
        }
        $department = int(rand()*60) + 1;

        printf ITEMS
("$i,\"$name\",%.2f,$weight,$discount,$department\n",
                $price);
}
```

Appendix G:

Generation Script for Customer Accounts and Customer Addresses

```
#!/usr/bin/perl -w
#
# customer.pl - generate customer_accounts and customer_addresses files
# simultaneously
#

use Time::Local;

our(@male_names, @male_values, @female_names, @female_values,
@last_names,
        @last_values, @city_names, @city_values, @states);

load_data();

open (CACCT, ">../raw/customer_accounts.csv");
open (CADDR, ">../raw/customer_addresses.csv");

for($i=1; $i<=1000000; $i++) {

        $n_addrs = int(rand()*5) + 1;
        $current_addr = int(rand()*$n_addrs) + 1;

        $ca = "$i," . get_random_name() . "," . generate_phone() . ","
                . $current_addr . "," . generate_balance() . ","
                . generate_dates() . "\n";
        print CACCT $ca;

        for($j=1;$j<=$n_addrs;$j++) {
                $caddr = "$i,$j," . generate_address() . ","
                        . get_random_city() . "\n";
                print CADDR $caddr;

        }
}


sub get_random_name {

        if(rand() < 0.52) {
                $first_index = rand() * 90.024;
                $first_name = select_name($first_index, \@female_names,
                        \@female_values);
        } else {
                $first_index = rand() * 90.040;
                $first_name = select_name($first_index, \@male_names,
                        \@male_values);
        }
        $last_index = rand() * 77.480;

        $last_name = select_name($last_index, \@last_names,
\@last_values);
```

```
        return "$first_name,$last_name";

} # get_random_name


sub select_name {

        my($index, $a, $b) = @_;

        @names = @{$a};
        @values = @{$b};

        for(my $i = 0; $i < $#names + 1; $i++) {

                if($index < $values[$i]) {
                        return $names[$i];
                }
        }
        print "Failed to find name!!\n";

} # select_name


sub get_random_city {

        $zip = sprintf "%05d", int (rand() * 100000);

        $index = rand();

        for(my $i = 0; $i < $#city_names + 1; $i++) {

                if($index < $city_values[$i]) {
                        return "$city_names[$i],$states[$i],$zip";
                }
        }
        print "Failed to find a city!  How is that possible?!\n";

} # get_random_city


sub generate_phone {

        return sprintf("%03d-%03d-%04d", int (rand() * 700) + 200,
                int(rand()*1000), int(rand()*10000));

} # generate_phone


sub generate_address {

        $house = int (rand() * 10000) + 1;
        $street = int (rand() * 150) + 1;
        if($street % 10 == 1) {
                $street .= "st";
        } elsif($street % 10 == 2) {
                $street .= "nd";
        } elsif($street % 10 == 3) {
                $street .= "rd";
        } else {
                $street .= "th";
        }

        $d = int(rand() * 20)+1;
        if($d > 18) {
                $dir = "East ";
        } elsif($d > 16) {
                $dir = "West ";
        } elsif($d > 14) {
                $dir = "South ";
        } elsif($d > 12) {
                $dir = "North ";
        } elsif($d > 11) {
                $dir = "NE ";
        } elsif($d > 10) {
                $dir = "NW ";
        } elsif($d > 9) {
                $dir = "SE ";
        } elsif($d > 8) {
                $dir = "SW ";
        } else {
                $dir = "";
        }

        $s = int(rand() * 4);
        $str = "Road" if $s == 0;
        $str = "Street" if $s == 1;
        $str = "Avenue" if $s == 2;
        $str = "Boulevard" if $s == 3;

        return "$house $dir$street $str";

} # generate_address


sub generate_balance {

        if(rand() < 0.95) {
                return "0.00";
        } else {
                return sprintf("%.2f", rand()*2000);
        }

} # generate_balance


sub generate_dates {

        # create date = 1/1/2003 + random in 5 yr
        # activity date = 7/1/2007 + random in 6 mo and > create date

        $y5start = timelocal(0,0,0,1,0,103);
        $y5end = timelocal(0,0,0,1,0,108);
        $m6start = timelocal(0,0,0,1,6,107);
        $cd = int(rand()*($y5end - $y5start)) + $y5start;

        if($cd > $m6start) {
                $diff = ($y5end - $cd);
                $ad = int(rand()*$diff) + $cd;
        } else {
                $diff = ($y5end - $m6start);
                $ad = int(rand()*$diff) + $m6start;
        }

        return '"' . my_format($cd) . '","' . my_format($ad) . '"';

} # generate_dates

sub my_format() {

        $d = $_[0];

        ($sec,$min,$hr,$day,$mon,$yr) = localtime($d);

        return sprintf("%02d/%02d/%4d %02d:%02d:%02d", $mon+1, $day,
$yr+1900,
                $hr, $min, $sec);

} # my_format

sub load_data {

        my $i = 0;
        open FILE, "<male.txt";
        while (<FILE>) {
                chop;
                ($male_names[$i], $male_values[$i]) = split /,/;
                $i++;
        }
        close FILE;

        $i = 0;
        open FILE, "<female.txt";
        while (<FILE>) {
                chop;
                ($female_names[$i], $female_values[$i]) = split /,/;
                $i++;
        }
        close FILE;

        $i = 0;
        open FILE, "<lastnames.txt";
        while (<FILE>) {
                chop;
                ($last_names[$i], $last_values[$i]) = split /,/;
                $i++;
        }
        close FILE;

        $i = 0;
        open FILE, "<cities.txt";
        while (<FILE>) {
                chop;
                ($city_values[$i], $states[$i], $city_names[$i]) =
split /,/;
                $i++;
        }
        close FILE;

} # load_data
```

Appendix H:

Generation Script for Item Ratings

```
#!/usr/bin/perl -w
#
# item_ratings.pl
#

use Time::Local;

$n_items = 25000; # from of the items.csv file
$n_cacct = 1000000; # from the customer_accounts file

open(CACCT,"<../raw/customer_accounts.csv");
while(<CACCT>) {
        @values = split /,/;
        $id = $values[0];
        $values[6] =~ /"(.*)"/;
        $cd[$id] = $1;
}
close(CACCT);

open(IRATING,">../raw/item_ratings.csv");

for($i=1;$i<=$n_items;$i++) {
        $d = rand();

# we double the below ratings and then they'll be halfed back when we
verify
# the rating date > create date. This means we need to read in the
create
# dates first.

        if($d < 0.11) {
                $n_ratings = 0;
        } elsif($d < 0.89) {
                $n_ratings = int(rand()*10*2) + 1;
        } elsif($d < 0.99) {
                $n_ratings = int(rand()*100*2) + 1;
        } else {
                $n_ratings = int(rand()*1000*2) + 1;
        }

        $j = 1;
        undef %cr;
        while($j<=$n_ratings) {
                $ca_id = int(rand()*$n_cacct) + 1;

                # skip if we did this customer already
                next if($cr{$ca_id});

                # this one counts
                $j++;
                $cr{$ca_id}=1;
                # rating date = 1/1/2003 + random in 5 yr
                $y5start = timelocal(0,0,0,1,0,103);
                $y5end = timelocal(0,0,0,1,0,108);
                $rd = int(rand()*($y5end - $y5start)) + $y5start;

                # reverse the create date
                ($mdy,$hms) = split / /, $cd[$ca_id];
                ($mon,$day,$yr) = split /\//,$mdy;
                ($hr,$min,$sec) = split /:/,$hms;
                $this_cd = timelocal(0+$sec, 0+$min, 0+$hr, 0+$day,
                        0+$mon-1,$yr-1900);

                # skip if create date after rating date
                next if $rd <= $this_cd;

                $rating = int(rand()*5) + 1;
                ($sec,$min,$hr,$day,$mon,$yr) = localtime($rd);
                $mon++; $yr+=1900;
                $rdstring = "\"$mon/$day/$yr $hr:$min:$sec\"";

                print IRATING "$i,$ca_id,$rating,$rdstring\n";

        }

}
```

Appendix I:

Generation Script for Store Inventories

```
#!/usr/bin/perl -w
#
# store_inventories.pl
#

$n_items = 25000; # from of the items.csv file
$n_stores = 250;

open(ITEMS,"<../raw/items.csv");
while(<ITEMS>) {
        @values = split /,/;
        $id = $values[0];
        $wholesale_price[$id] = $values[2];
}
close(ITEMS);

open(SINV,">../raw/store_inventory.csv");

for($i=1;$i<=$n_stores;$i++) {
  for($j=1;$j<=$n_items;$j++) {

        # 40% chance skip, else 1-6 qty.
        if(rand() < 0.4) {
                next;
        } else {
                $qty = int(rand()*6) + 1;
        }

        # retail = wholesale plus 5-40% markup
        $markup = 0.05 * (int(rand()*8) + 1);
        $retail_price = $wholesale_price[$j] * (1 + $markup);

        printf SINV ("$i,$j,$qty,%.2f\n", $retail_price);
  }
}
```

Appendix J:

Generation Script for Volume Discounts

```
#!/usr/bin/perl -w
#
# volume_discounts.pl
#

open(VOLD,">../raw/volume_discounts.csv");
print VOLD "100,0.00\n";
$value = 100;
for($i=1;$i<=10;$i++) {
        $value = 2 * $value;
        printf VOLD ("%d,%.2f\n", $value, 0.01 * $i);
}
```

Appendix K:

Generation Script for Club Members

```
#!/usr/bin/perl -w
#
# club.pl
#

open(CLUB1,">../raw/club1.csv");

for($i=1;$i<=1000000;$i++) {
        # 15% of accounts are club members
        if(rand() < 0.15) {
                $d = rand();
                # Three levels of membership with different discounts
                if($d <0.75) {
                        $disc = 0.05;
                } elsif($d < 0.95) {
                        $disc = 0.10;
                } else {
                        $disc = 0.15;
                }
                print CLUB1 "$i,$disc\n";
        }
}
```

Appendix L:

Generation Script for Transactions, Transaction Items, and Club Members Tables

```
#!/usr/bin/perl -w
#
# tran.pl - generates transactions, transaction_items, and club_members
#

use Time::Local;

our (%shipwts, %shipcosts, @vold_prices, @vold_discounts,
@club_discount,
        @club_total_qty, @club_total_spent, @club_last_purchase_date,
        @create_date, @cust_state, @item_price, @item_weight,
@store_state,
        %tax_rates);

$n_trans = 30000000; # actual number will be half this by date check

$n_items = 25000; # from of the items.csv file

our $n_cacct = 1000000; # from the customer_accounts file

$y5start = timelocal(0,0,0,1,0,103);
$y5end = timelocal(0,0,0,1,0,108);

&load_shipping;
&load_volume_discounts;
&load_tax;
&load_club_discounts;
&load_customer_data;
&load_stores;
&load_items;

open (TRANS, ">../raw/transactions.csv");
open (TITEMS, ">../raw/transaction_items.csv");

for($tid=1;$tid<=$n_trans;$tid++) {

        $cid = int(rand()*$n_cacct) + 1;

        $to_state = $customer_state[$cid];

        # tran date = 1/1/2003 + random in 5 yr
        $td = int(rand()*($y5end - $y5start)) + $y5start;

        # skip if tran date is before customer create date
        next if($td < $create_date[$cid]);

        $store_id = int(rand()*250) + 1;
        $from_state = $store_state[$store_id];

        # transaction items: 1-3 (75%) or 1-19 (25%)
        if(rand() < 0.75) {
                $n_trans_items = int(rand()*3) + 1;
        } else {
                $n_trans_items = int(rand()*19) + 1;
        }

        $subtotal = $total_wt = 0;
        for($seqno=1; $seqno<=$n_trans_items; $seqno++) {
                $item_id = int(rand()*$n_items) + 1;
                $price = $item_price[$item_id];
                $markup = 0.80 + 0.01 * int(rand()*60);
                $price = money_round($price * $markup);

                # qty = 1 (90%) or 1-4 (10%)
                if(rand() < 0.9) {
                        $qty  = 1;
                } else {
                        $qty = int(rand()*4) + 1;
                }
                $extended_price = $price * $qty;
                $weight = $item_weight[$item_id];
                if(rand() < 0.5) {
                        $discount = 0.0;
                } else {
                        $discount = 0.01 * int(rand()*40);
                }
                $discounted_price = money_round($price * (1.0 -
$discount));
                $total_wt += $weight * $qty;
                $subtotal += $discounted_price;

                print TITEMS "$tid,$seqno,$item_id,$price,$qty,"
                        .
"$extended_price,$discount,$discounted_price\n";

                $club_total_qty[$cid] += $qty;
        }
        $voldisc = compute_volume_discount($subtotal);
        $club_disc = $club_discount[$cid];

        $disc_subtotal = money_round($subtotal * (1.0-$voldisc-
$club_disc));
        $shipping =  compute_shipping($from_state, $to_state,
$total_wt);
        $tax = $tax_rates{$to_state};
        $total = money_round($disc_subtotal +
                $shipping + 0.01*$tax*$disc_subtotal);

        $club_total_spent[$cid] += $total;
        if($td > $club_last_purchase_date[$cid]) {
                $club_last_purchase_date[$cid] = $td;
        }
        $date = pretty_date($td);
        print TRANS
"$tid,$cid,$store_id,\"$date\",$subtotal,$total_wt,"
                . "$club_disc,$voldisc,$shipping,$tax,$total\n";
} # loop over trans
```

```
close(TRANS);
close(TITEMS);

open(CLUB2, ">../raw/club2.csv");
for($cid=1;$cid<=$n_cacct;$cid++) {

        # member date = 1/1/2003 + random in 5 yr
        $md = int(rand()*($y5end - $y5start)) + $y5start;

        # skip if tran date is before customer create date
        if($md < $create_date[$cid]) {
                $md = $create_date[$cid];
        }
        $member_date = pretty_date($md);
        if($club_last_purchase_date[$cid]) {
                $clpd = pretty_date($club_last_purchase_date[$cid]);
        } else {
                $clpd = "";
        }

        if($club_discount[$cid] != 0) {
          print CLUB2
"$cid,$club_total_qty[$cid],$club_total_spent[$cid],"
                . "\"$member_date\",\"$clpd\","
                . "$club_discount[$cid]\n";
        }
}
close(CLUB2);

sub load_shipping {
        open(SHIP, "../raw/shipping.csv");
        while(<SHIP>) {
                chomp;
                ($from,$to,$wt,$cost) = split /,/;
                $key = "$from,$to";
                $key =~ /"(.*)","(.*)"/;
                $key = "$1,$2";
                $shipwts{$key} .= $wt . ",";
                $shipcosts{$key} .= $cost . ",";
        }
        close(SHIP);
}

sub compute_shipping {
        ($from,$to,$this_wt) = @_;
        $key = "$from,$to";
        @wts = split /,/,$shipwts{$key};
        @costs = split /,/,$shipcosts{$key};

        for($k=0;$k<=$#wts;$k++) {
                $this_cost = $costs[$k];
                last if($this_wt <= $wts[$k]);
        }
        return $this_cost;
}

sub load_volume_discounts {
        open(VOLD, "../raw/volume_discounts.csv");
        $k=0;
        while(<VOLD>) {
                chomp;
                ($vold_prices[$k],$vold_discounts[$k]) = split /,/;
                $k++;
        }
        close(VOLD);
}

sub compute_volume_discount {
        $total_price = shift;

        for($k=0;$k<=$#vold_prices;$k++) {
                $this_discount = $vold_discounts[$k];
                last if($total_price < $vold_prices[$k]);
        }
        return $this_discount;
}

sub load_tax {
        open(TAX, "../raw/states.csv");
        while(<TAX>) {
                chomp;
                ($state,$tax) = split /,/;
                $state =~ /"(.*)"/;
                $tax_rates{$1} = $tax;
        }
        close(TAX);
}


sub load_club_discounts {
        for($i=1;$i<=$n_cacct;$i++) {
                $club_discount[$i] = 0;
                $club_total_qty[$i] = 0;
                $club_total_spent[$i] = 0;
                $club_last_purchase_date[$i] = 0;
        }
        open(CLUB1, "<../raw/club1.csv");
        while(<CLUB1>) {
                chomp;
                ($cid, $disc) = split /,/;
                last if($cid > $n_cacct);
                $club_discount[$cid] = $disc;
        }
        close(CLUB1);
}

sub load_customer_data {
        my(@addr);

        open(CACCT,"<../raw/customer_accounts.csv");
        $k = 0;
        while(<CACCT>) {
                chomp;
                @values = split /,/;
                $cid = $values[0];
                last if ($cid > $n_cacct);

                $addr[$cid] = $values[4];

                # reverse the create date
                $values[6] =~ /"(.*)"/;
                ($mdy,$hms) = split / /, $1;
                ($mon,$day,$yr) = split /\//,$mdy;
                ($hr,$min,$sec) = split /:/,$hms;
                $create_date[$cid] = timelocal(0+$sec, 0+$min, 0+$hr,
0+$day,
                        0+$mon-1,$yr-1900);
        }
        close(CACCT);

        open(CADDR,"<../raw/customer_addresses.csv");
        while(<CADDR>) {
                @values = split /,/;
                $cid = $values[0];
                $seq_no = $values[1];
                $state = $values[4];
                last if($cid > $n_cacct);
                if($addr[$cid] == $seq_no) {
                        $customer_state[$cid] = $state;
                }
        }
        close(CADDR);
}

sub load_stores {
        open(STORES, "<../raw/stores.csv");
        while(<STORES>) {
                chomp;
                @values = split /,/;
                $sid = $values[0];
                $store_state[$sid] = $values[3];
        }
        close(STORES);
}

sub load_items {
        open(ITEMS, "<../raw/items.csv");
        while(<ITEMS>) {
                chomp;
                @values = split /,/;
                $item_id = $values[0];
                $item_price[$item_id] = $values[2];
                $item_weight[$item_id] = $values[3];
        }
        close(ITEMS);
}

sub pretty_date {
        $date = shift;
        ($sec,$min,$hr,$day,$mon,$yr) = localtime($date);
        $mon++; $yr+=1900;
        return sprintf("%02d/%02d/%4d %02d:%02d:%02d", $mon, $day, $yr,
                $hr, $min, $sec);
}

sub money_round {
        $value = shift;
        return int($value*100+0.5)/100.0;
}
```

Appendix M:

Benchmark Script for Batch Load

```
#!/usr/bin/perl -w
#
# loadb.pl - Batch load of benchmark raw data files into a database
#

$dbtype = "DB2";
#$dbtype = "Firebird";
#$dbtype = "Sybase";
#$dbtype = "Postgresql";
#$dbtype = "Oracle";
#$dbtype = "MySQL";

use Time::HiRes qw(gettimeofday tv_interval);   # High resolution
timing
use Switch;                         # For switch/case statements
use DBI;                            # General database interface

our($dbh, $dsn);

switch($dbtype) {
      case "Sybase" {
            # use DBD::Sybase;            # Sybase specific interface
            $dsn = "DBI:Sybase:server=VADER";
            $db_user="bench";
            $db_pass="benchpw";
            $endsql = "";
      }
      case "Postgresql" {
            # use DBD::Pg;                # Postgresql specific
interface
            $dsn = "DBI:Pg:";
            $db_user="bench";
            $db_pass="bench";
            $endsql = ";";
      }
      case "Oracle" {
            #use DBD::Oracle;             # Oracle specific interface
            $dsn = "DBI:Oracle:";
            $db_user="bench";
            $db_pass="bench";
            $endsql = "";
      }
      case "MySQL" {
            #use DBD::mysql;        # MySQL specific interface
            $dsn = "DBI:mysql:database=bench;host=localhost;port=3306";
            $db_user="bench_user";
            $db_pass="bench1";
            $endsql = ";";
      }
      case "Firebird" {
            use DBD::InterBase;           # Firebird specific interface
            $dsn =
"DBI:InterBase:db=/opt/firebird/bench.fdb;ib_dialect=3";
            $db_user="bench";
            $db_pass="bench";
            $endsql = "";
      }
      case "DB2" {
            use DBD::DB2;
            use DBD::DB2::Constants;
            $dsn = "dbi:DB2:bench";
            $db_user = "";
            $db_user = "";
            $endsql = "";
      }
      default { die "DBD $dbtype not found."; }
}

our(@table, @placeholders, @fields, @date_fields);
&table_definitions;


$n_children = 14;


# Empty out all of the tables.
#
$trun_dbh = DBI->connect($dsn, $db_user, $db_pass);

for($i=0; $i<$n_children; $i++) {
      $sql = "truncate table $table[$i]$endsql";
      if($dbtype eq "Firebird") {
            $sql = "delete from $table[$i]";
      }
      if($dbtype eq "DB2") {
            $sql = "alter table $table[$i] activate not logged "
                  . "initially with empty table";
      }
      print($sql . "\n");
      $trun_dbh->do($sql);
}
$trun_dbh->disconnect;


$start_time = [gettimeofday];             # Start timer

# Run each child process in its own thread, for parallelism.
#
for($i=0; $i<$n_children; $i++) {
      $pid = fork();
      child_proc($i) if(!$pid);
}

$children_done = 0;
$child = 0;
while ($child != -1) {
      $child = wait();  # Returns -1 when no more children waiting.
      if($child != -1) {
            $children_done++;
      }
}

$elapsed_time = tv_interval($start_time); # End timer

print "$children_done finished $elapsed_time sec.\n";


# End of script.


sub child_proc {
      $child_no = shift;
      print "Running child number $child_no\n";

      # Login and create database handler
      #
      $dbh = DBI->connect($dsn, $db_user, $db_pass);

      if($dbtype eq "Oracle") {
            $dbh->do("alter session set " .
                  "nls_date_format='yyyy-mm-dd hh24:mi:ss'");
      }

      load_table($table[$child_no], $placeholders[$child_no],
            $fields[$child_no], $date_fields[$child_no]);
      exit 0;
}


sub load_table() {
      $table = shift;
      $placeholder = shift;
      $fields = shift;
      $date_cols = shift;

      my $sth;

      if($dbtype eq "Sybase") {
            if($table eq "customer_accounts" or $table eq "stores" or
              $table eq "items" or $table eq "transactions" or
              $table eq "departments") {
                  $presql = "set identity_insert $table on";
                  $sth = $dbh->prepare($presql);
                  $sth->execute;
            }
      }
      $sql = "insert into $table ($fields) values
($placeholder)$endsql";
      $sth = $dbh->prepare($sql);

      $file = "./raw/$table.csv\n";
      open RAWFILE, "<$file" or die "Can't open file: $file";
      while(<RAWFILE>) {
            chomp;            # remove newline
            s/"//g;           # remove quotes
            $values = fix_dates($_, $date_cols);
            @value_array = split /,/, $values;
            if($dbtype eq "Postgresql" or $dbtype eq "Firebird"
              or $dbtype eq "DB2") {
                  @value_array = replace_undef(@value_array);
            }
            $sth->execute(@value_array);
      }
      $sth->finish;
}


# Convert date to MySQL format.
#
sub mysql_convert_date {
      $old_date = shift;
      return "null" if $old_date eq '""';
      return $old_date;
}


# Convert date to Oracle format.
#
sub oracle_convert_date {
      $old_date = shift;
      return "null" if $old_date eq '""';
      return $old_date;
# We used alter session nls_date_format at the beginning.  Otherwise we
# would need to recode to use:
#     to_date('$old_date', 'yyyy-mm-dd hh24:mi:ss')
}


# Convert date to Postgresql format.
#
sub pg_convert_date {
      $old_date = shift;
      return "null" if $old_date eq '';
      return $old_date;
}


# Convert date to DB2 format.
#
sub db2_convert_date {
      $old_date = shift;
      return "null" if $old_date eq '';
      return $old_date;
}


# Convert date to Sybase format.
#
sub sybase_convert_date {
      $old_date = shift;
      return "null" if $old_date eq '""';
      return $old_date;
```

```
}


# Convert date to Firebird format.
#
sub firebird_convert_date {
	$old_date = shift;
	return "null" if $old_date eq '';
	return $old_date;
}


# For postgresql: replace "null" with undef in an array.
#
sub replace_undef {
	@foo = @_;
	for($i=0; $i<= $#foo; $i++) {
		if($foo[$i] eq "null") {
			$foo[$i] = undef;
		}
	}
	return @foo;
}


# Identify which columns need date conversion and fix them.
#
sub fix_dates {
	my @cols = split /,/, shift;
	my @nums = split /,/, shift;
	switch($dbtype) {
		case "Sybase" {
			foreach $key (@nums) {
				$cols[$key] = sybase_convert_date($cols[$key]);
			}
		}
		case "Postgresql" {
			foreach $key (@nums) {
				$cols[$key] = pg_convert_date($cols[$key]);
			}
		}
		case "DB2" {
			foreach $key (@nums) {
				$cols[$key] = db2_convert_date($cols[$key]);
			}
		}
		case "Oracle" {
			foreach $key (@nums) {
				$cols[$key] = oracle_convert_date($cols[$key]);
			}
		}
		case "MySQL" {
			foreach $key (@nums) {
				$cols[$key] = mysql_convert_date($cols[$key]);
			}
		}
		case "Firebird" {
```

```
                  foreach $key (@nums) {
                        $cols[$key] =
firebird_convert_date($cols[$key]);
                  }
            }
            default { die "DBD $dbtype"; }
      }
      return(join ",", @cols);
}



sub table_definitions {

      $table[0] = "club_members";
      $placeholders[0] = "?, ?, ?, ?, ?, ?";
      $fields[0] = "customer_id, total_quantity, total_spent, "
            . "member_since, last_purchase_date, discount";
      $date_fields[0] = "3,4";

      $table[1] = "customer_accounts";
      $placeholders[1] = "?, ?, ?, ?, ?, ?, ?, ?";
      $fields[1] = "customer_id, first_name, last_name, phone, "
            . "current_address, balance, creation_date, activity_date";
      $date_fields[1] = "6,7";

      $table[2] = "customer_addresses";
      $placeholders[2] = "?, ?, ?, ?, ?, ?";
      $fields[2] = "customer_id, sequence_number, street_address, city,
"
            . "state, zip";
      $date_fields[2] = "";

      $table[3] = "department_discounts";
      $placeholders[3] = "?, ?, ?";
      $fields[3] = "store_id, department_id, sale_discount";
      $date_fields[3] = "";

      $table[4] = "departments";
      $placeholders[4] = "?, ?";
      $fields[4] = "department_id, name";
      $date_fields[4] = "";

      $table[5] = "item_ratings";
      $placeholders[5] = "?, ?, ?, ?";
      $fields[5] = "item_id, customer_id, rating, date_updated";
      $date_fields[5] = "3";

      $table[6] = "items";
      $placeholders[6] = "?, ?, ?, ?, ?, ?";
      $fields[6] = "item_id, name, wholesale_price, weight, "
            . "item_discount, department_id";
      $date_fields[6] = "";

      $table[7] = "shipping";
      $placeholders[7] = "?, ?, ?, ?";
      $fields[7] = "from_state, to_state, weight, shipping_cost";
      $date_fields[7] = "";

      $table[8] = "states";
      $placeholders[8] = "?, ?";
      $fields[8] = "state, tax_rate";
      $date_fields[8] = "";

      $table[9] = "store_inventories";
      $placeholders[9] = "?, ?, ?, ?";
      $fields[9] = "store_id, item_id, quantity, retail_price";
      $date_fields[9] = "";

      $table[10] = "stores";
      $placeholders[10] = "?, ?, ?, ?";
      $fields[10] = "store_id, store_name, store_discount,
ships_from_state";
      $date_fields[10] = "";

      $table[11} = "transaction_items";
      $placeholders[11] = "?, ?, ?, ?, ?, ?, ?, ?";
      $fields[11] = "transaction_id, sequence_number, item_id, price, "
            . "quantity, extended_price, discount, discounted_price";
      $date_fields[11] = "";

      $table[12] = "transactions";
      $placeholders[12] = "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?";
      $fields[12] = "transaction_id, customer_id, store_id, tran_date,
"
            . "subtotal, total_weight, club_discount, volume_discount,
"
            . "shipping_cost, taxes, total";
      $date_fields[12] = "3";

      $table[13} = "volume_discounts";
      $placeholders[13] = "?, ?";
      $fields[13] = "total_purchase, discount";
      $date_fields[13] = "";

}
```

Appendix N:

Benchmark Script for Transaction Processing

```
#!/usr/bin/perl -w
#
# tranb.pl - run transactions as part of the benchmark case
#

$dbtype = "DB2";
#$dbtype = "Sybase";
#$dbtype = "Postgresql";
#$dbtype = "Oracle";
#$dbtype = "MySQL";

use Time::HiRes qw(gettimeofday tv_interval);   # High resolution
timing
use Time::Local;                    # reverse of localtime
use Switch;                         # For switch/case statements
use DBI;                            # General database interface
#use DBD::InterBase;          # Firebird specific interface
#use DBD::Sybase;       # Sybase specific interface
#use DBD::Pg;                 # Postgresql specific interface
use DBD::DB2;                 # DB2 specific interface
use DBD::DB2::Constants;      # more for DB2
#use DBD::Oracle;       # Oracle specific interface
#use DBD::mysql;        # MySQL specific interface


$| = 1;

our($dbh, $dsn);
our($endsql, $ca_seq, $stores_seq, $items_seq, $tran_seq, $dept_seq);
our($tablelockmode);

switch($dbtype) {
      case "Firebird" {
            $dsn =
"DBI:InterBase:db=/opt/firebird/bench.fdb;ib_dialect=3";
            $db_user="bench";
            $db_pass="bench";
            $endsql = "";
            $ca_seq = "gen_id(customer_accounts_seq,1),";
            $stores_seq = "gen_id(stores_seq,1),";
            $items_seq = "gen_id(items_seq,1),";
            $tran_seq = "gen_id(transactions_seq,1),";
            $dept_seq = "gen_id(departments_seq,1),";
            $tablelockmode = "with lock";
      }
      case "DB2" {
            $dsn = "dbi:DB2:bench";
            $db_user="";
            $db_pass="";
            $endsql = "";
            $ca_seq = $stores_seq = $items_seq =
            $tran_seq = $dept_seq= "default, ";
```

```
                $tablelockmode = "IN EXCLUSIVE MODE";
        }
        case "Sybase" {
                $dsn = "DBI:Sybase:server=VADER";
                $db_user="bench";
                $db_pass="benchpw";
                $endsql = "";
                $ca_seq = $stores_seq = $items_seq =
                $tran_seq = $dept_seq= "";
                $tablelockmode = "IN EXCLUSIVE MODE";
        }
        case "Postgresql" {
                $dsn = "DBI:Pg:";
                $db_user="bench";
                $db_pass="bench";
                $endsql = "";
                $ca_seq = "nextval(\"customer_accounts_seq\"),";
                $stores_seq = "nextval(\"stores_seq\"),";
                $items_seq = "nextval(\"items_seq\"),";
                $tran_seq = "nextval(\"transactions_seq\"),";
                $dept_seq = "nextval(\"departments_seq\"),";
                $tablelockmode = "IN EXCLUSIVE MODE";
        }
        case "Oracle" {
                $dsn = "DBI:Oracle:";
                $db_user="bench";
                $db_pass="bench";
                $endsql = "";
                $ca_seq = "customer_accounts_seq.nextval,";
                $stores_seq = "stores_seq.nextval,";
                $items_seq = "items_seq.nextval,";
                $tran_seq = "transactions_seq.nextval,";
                $dept_seq = "departments_seq.nextval,";
                $tablelockmode = "IN EXCLUSIVE MODE";
        }
        case "MySQL" {
                $dsn = "DBI:mysql:database=bench;host=localhost;port=3306";
                $db_user="bench_user";
                $db_pass="bench1";
                $endsql = ";";
                $ca_seq = $stores_seq = $items_seq =
                $tran_seq = $dept_seq= "null,";
                $tablelockmode = "WRITE";
        }
        default { die "DBD $dbtype not found."; }
}

our($n_children, $n_transactions_per_child);
$n_children = 20;       # 20 processes
$n_transactions_per_child = 500;
$n_transactions_per_child = 500;


our(@male_names, @male_values, @female_names, @female_values,
@last_names,
        @last_values, @city_names, @city_values, @states);
our($INITIAL_CUSTOMERS, $INITIAL_ITEMS,
      $INITIAL_STORES, $INITIAL_DEPARTMENTS);

load_data();      # Get sample names from text files, set constants

$start_time = [gettimeofday];             # Start timer

# Run each child process in its own thread, for parallelism.
#
for($i=0; $i<$n_children; $i++) {
      $pid = fork();
      child_proc($i) if(!$pid);
}

$children_done = 0;
$child = 0;
while ($child != -1) {
      $child = wait();  # Returns -1 when no more children waiting.
      if($child != -1) {
            $children_done++;
      }
}

$elapsed_time = tv_interval($start_time); # End timer

print "$children_done finished $elapsed_time sec.\n";


# End of script.


sub child_proc {
      $child_no = shift;
      # print "Running child number $child_no\n";

      # Login and create database handler
      #
      $dbh = DBI->connect($dsn, $db_user, $db_pass,
            {PrintError => 1, RaiseError => 1, AutoCommit => 0})
            or die "Database connection failed: $DBI::errstr";

      if($dbtype eq "Sybase") {
            $dbh->do("set arithabort numeric_truncation off");
      }
      if($dbtype eq "Oracle") {
            # Tell Oracle to use our date format
            $dbh->do("alter session set " .
                  "nls_date_format='mm/dd/yyyy hh24:mi:ss'");
            $dbh->commit();
      }

      run_transactions($child_no);

      exit 0;
}


# Run a series of transactions, selected randomly.
```

```
#
sub run_transactions {
      $child_no = shift;

      for($tran_no=0; $tran_no<$n_transactions_per_child; $tran_no++) {

            # print "Child $child_no tran $tran_no ";
            $x = rand();
            if($x < 0.70) {
                  # very common transaction, 60% chance
                  very_common_transaction();
            } elsif ($x < 0.96) {
                  # common transaction, 30% chance
                  common_transaction();
            } else {
                  # rare transaction, 10% chance
                  rare_transaction();
            }
      }

} # run_transactions


sub fix_one_date {
      $date = shift;
      switch($dbtype) {
            case "Sybase" {
                  # Sybase needs no conversion.
                  return $date;
            }
            case "MySQL" {
                  # MySQL needs no conversion.
                  return $date;
            }
            case "DB2" {
                  # DB2 needs to convert dates
                  return db2_convert_date($date);
            }
            case "Oracle" {
                  # Oracle needs no conversion.
                  return $date;
            }
            case "Postgresql" {
                  # Postgresql needs no conversion.
                  return $date;
            }
            case "Firebird" {
                  # Firebird needs no conversion.
                  return $date;
            }
            default { die "DBD $dbtype"; }
      }
} # fix_one_date


sub very_common_transaction {
      $key = int(rand()*6);
      $tran_start_time = [gettimeofday];          # Start timer

      switch($key) {
            case 0 {
                  create_customer();
            }
            case 1 {
                  update_customer_balance();
            }
            case 2 {
                  customer_rates_an_item();
            }
            case 3 {
                  customer_purchases_item();
            }
            case 4 {
                  adjust_inventory();
            }
            case 5 {
                  adjust_item_for_all_stores();
            }
      }
      $tran_elapsed_time = tv_interval($tran_start_time);    # End timer
      # print "very_common $key $tran_elapsed_time\n";

}


sub common_transaction {

      $key = int(rand()*6);
      $tran_start_time = [gettimeofday];          # Start timer

      switch($key) {
            case 0 {
                  update_phone();
            }
            case 1 {
                  update_address();
            }
            case 2 {
                  add_new_address();
            }
            case 3 {
                  change_current_address();
            }
            case 4 {
                  customer_joins_club();
            }
            case 5 {
                  create_new_item();
            }
      }
      $tran_elapsed_time = tv_interval($tran_start_time);    # End timer
      # print "common $key $tran_elapsed_time\n";
```

```
}


sub rare_transaction {

	$key = int(rand()*7);
	$tran_start_time = [gettimeofday];		# Start timer

	switch($key) {
		case 0 {
			create_new_store();
		}
		case 1 {
			close_store();
		}
		case 2 {
			update_store_discount();
		}
		case 3 {
			adjust_department_discount();
		}
		case 4 {
			update_volume_discount();
		}
		case 5 {
			update_shipping();
		}
		case 6 {
			update_item();
		}
	}
	$tran_elapsed_time = tv_interval($tran_start_time);	# End timer
	# print "rare $key $tran_elapsed_time\n";

}


sub create_customer {

	($sec,$min,$hr,undef,undef,undef) = localtime(time);
	# Put timestamp in 1/1/08
	$now = timelocal($sec,$min,$hr,1,0,2008);
	# Convert to general format
	$now = '"' . my_format($now) . '"';
	# Convert to DB-specific format
	$now = fix_one_date($now);

	# start tran - drops out of eval block on error
	eval {
		# insert customer account record
		$sql = "INSERT INTO customer_accounts VALUES($ca_seq "
			. get_random_name() . "," . generate_phone()
			. ",1,0.00,$now,$now)";
		$sql = fix_quotes($sql);
		$sth = $dbh->prepare($sql);
		$sth->execute();
```

```
            # get the insert id from the auto_increment
            switch($dbtype) {
              case "MySQL" {
                  $customer_id = $dbh->last_insert_id(undef,undef,
                  "customer_accounts","customer_id");
              }
              case "Oracle" {
                  $customer_id =
                        get_oracle_sequence("customer_accounts_seq");
              }
              case "DB2" {
                  $customer_id = get_db2_last_insert_id();
              }
              case "Postgresql" {
                  $customer_id =
                        get_pg_sequence("customer_accounts_seq");
              }
              case "Sybase" {
                  $customer_id = $dbh->last_insert_id(undef,undef,
                  "customer_accounts","customer_id");
              }
              case "Firebird" {
                  $customer_id =
                        get_firebird_sequence("customer_accounts_seq");
              }
              default { die "Unknown dbtype: $dbtype"; }
            }

            # insert customer address record
            $sql = "INSERT INTO customer_addresses VALUES($customer_id,
1,"
                  . generate_address() . "," . get_random_city() . ")";
            $sql = fix_quotes($sql);
            $sth = $dbh->prepare($sql);
            $sth->execute();

            # commit tran
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # create_customer


sub update_customer_balance {

      # select a customer account randomly
      $id = int(rand()*($INITIAL_CUSTOMERS-10));

      # start tran - drops out of eval block on error
      eval {
            # find a valid id, in case we picked a missing record
            $sql = "SELECT MIN(customer_id) AS \"THIS_ID\" "
                  . "FROM customer_accounts "
                  . "WHERE customer_id > $id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $customer_id = $row{THIS_ID};

            $sth->finish();

            # subtract 100 from customer balance
            $sql = "UPDATE customer_accounts SET balance = balance -
100.0 "
                  . "WHERE customer_id = $customer_id";
            $dbh->do($sql);

            # commit tran
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # update_customer_balance


sub customer_rates_an_item {

      # select a customer account and item randomly
      $customer_id = int(rand()*($INITIAL_CUSTOMERS-10));
      $item_id = int(rand()*($INITIAL_ITEMS-10));

      # current timestamp
      ($sec,$min,$hr,undef,undef,undef) = localtime(time);
      # Put timestamp in 1/1/08
      $now = timelocal($sec,$min,$hr,1,0,2008);
      # Convert to general format
      $now = '"' . my_format($now) . '"';
      # Convert to DB-specific format
      $now = fix_one_date($now);

      # new rating
      $rating = int(rand()*5)+1;

      # start tran - drops out of eval block on error
      eval {
            # find a valid customer id, in case we picked a missing
record
            $sql = "SELECT MIN(customer_id) AS \"THIS_ID\" "
                  . "FROM customer_accounts "
                  . "WHERE customer_id > $customer_id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $customer_id = $row{THIS_ID};
            $sth->finish();
            # find a valid item id, in case we picked a missing record
            $sql = "SELECT MIN(item_id) AS \"THIS_ID\" FROM items "
                  . "WHERE item_id > $item_id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $item_id = $row{THIS_ID};
            $sth->finish();

            # lock the item_ratings table
            if($dbtype eq "Sybase") {
                  $dbh->do("BEGIN TRAN");
            }
            if($dbtype ne "Firebird") {
                  $dbh->do("LOCK TABLE item_ratings $tablelockmode");
            }

            # see if this item's already been rating by this user
            if($dbtype eq "Firebird") {
                  # Firebird's locking is nonstandard
                  $sql  = "SELECT * FROM item_ratings WHERE item_id = "
                  . "$item_id and customer_id = $customer_id with
lock";
                  $sth = $dbh->prepare($sql);
                  $sth->execute();
                  $sth->finish();
            }
            $sql = "SELECT count(*) as \"CNT\" FROM item_ratings "
                  . "WHERE item_id = $item_id and "
                  . "customer_id = $customer_id";

            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $count = $row{CNT};
            $sth->finish();

            if($count == 0) {
                  # insert new rating
                  $sql = "INSERT INTO item_ratings VALUES($item_id, "
                        . "$customer_id, $rating, $now)";
                  $sql = fix_quotes($sql);
                  $dbh->do($sql);
            } else {
                  # update with new rating
                  $sql = "UPDATE item_ratings set rating = $rating, "
                        . "date_updated = $now "
                        . "WHERE item_id = $item_id "
                        . "AND customer_id = $customer_id";
                  $sql = fix_quotes($sql);
                  $dbh->do($sql);
            }

            # commit tran
            if($dbtype eq "Sybase") {
                  $dbh->do("COMMIT TRAN");
            }
            if($dbtype eq "MySQL") {
                  $dbh->do("UNLOCK TABLES");
            }
            if($dbtype eq "Firebird") {
                  $dbh->do("COMMIT");
            }
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            if($dbtype eq "Sybase") {
                  $dbh->do("ROLLBACK TRAN");
            }
            if($dbtype eq "MySQL") {
                  $dbh->do("UNLOCK TABLES");
            }
            if($dbtype eq "Firebird") {
                  $dbh->do("ROLLBACK");
            }
            $dbh->rollback(); # dies if rollback fails
      }

} # customer_rates_an_item


sub customer_purchases_item {

      # select a customer account and item randomly
      $customer_id = int(rand()*($INITIAL_CUSTOMERS-10));

      # current timestamp (for hour, minute, second)
      ($sec,$min,$hr,undef,undef,undef) = localtime(time);
      # set date part to 1/1/08
      $now = timelocal($sec,$min,$hr,1,0,2008);
      # Convert to general format
      $now = '"' . my_format($now) . '"';
      # Convert to DB-specific format
      $now = fix_one_date($now);

      # find a valid customer id, in case we picked a missing record
      # and then get customer's tax rate, ship-to state,
      # and club member discount
      $sql = "SELECT c.customer_id AS \"CUSTOMER_ID\", "
            . "ca.state AS \"STATE\", "
            . "COALESCE(cm.discount, 0) as \"DISCOUNT\", "
            . "s.tax_rate as \"TAX\" "
            . "FROM customer_accounts c "
            . "INNER JOIN customer_addresses ca "
            . "ON c.customer_id = ca.customer_id "
            . "INNER JOIN states s ON ca.state = s.state "
            . "LEFT OUTER JOIN club_members cm "
            . "ON cm.customer_id = c.customer_id "
            . "WHERE c.current_address = ca.sequence_number "
            . "AND c.customer_id = "
            . "(SELECT MIN(customer_id) FROM customer_accounts "
            . "WHERE customer_id > $customer_id)";
$sth = $dbh->prepare($sql);
$sth->execute();
$rowref = $sth->fetchrow_hashref;
  %row = %$rowref;
$customer_id = $row{CUSTOMER_ID};
$to_state = $row{STATE};
$club_discount = $row{DISCOUNT};
$tax = $row{TAX};
$sth->finish();
$dbh->commit();         # avoid locks

# find the max item id
$sql = "SELECT max(item_id) as \"ITEM_ID\" FROM items";
$sth = $dbh->prepare($sql);
$sth->execute();
$rowref = $sth->fetchrow_hashref;
%row = %$rowref;
$max_item_id = $row{ITEM_ID};
$sth->finish();
$max_item_id = 25000 if($dbtype eq "Sybase");

# find the max store id
$sql = "SELECT max(store_id) as \"STORE_ID\" FROM stores";
$sth = $dbh->prepare($sql);
$sth->execute();
$rowref = $sth->fetchrow_hashref;
%row = %$rowref;
$max_store_id = $row{STORE_ID};
$sth->finish();
$max_store_id = 250 if($dbtype eq "Sybase");

# pick a random store, get location
$store_id = int(rand()*($max_store_id-10));
$sql = "SELECT store_id AS \"THIS_ID\", "
      . "ships_from_state AS \"FROM_STATE\" "
      . "FROM stores WHERE store_id = "
      . "(SELECT MIN(store_id) FROM stores "
      .  "WHERE store_id >= $store_id)";

$sth = $dbh->prepare($sql);
$sth->execute();
$rowref = $sth->fetchrow_hashref;
%row = %$rowref;
$store_id = $row{THIS_ID};
$from_state = $row{FROM_STATE};
$sth->finish();
$dbh->commit();         # avoid locks

# select a random number of items similar to the gen script
if(rand() < 0.75) {     # 1-3 (75%) or 1-19 (25%)
      $n_trans_items = int(rand()*3) + 1;
} else {
      $n_trans_items = int(rand()*19) + 1;
}
```

```
      # loop over the items - get item from store inventory, and its
data
      # decrement store inventory
      $total_weight = 0;
      $subtotal = 0;
      $total_price = 0;
      for($i=0; $i<$n_trans_items; $i++) {
            $my_item_id = int(rand()*($max_item_id-100));
            $sql = "
SELECT s.store_discount AS \"STORE_DISCOUNT\",
  s.ships_from_state AS \"FROM_STATE\",
  si.retail_price AS \"PRICE\",
  COALESCE(dd.sale_discount, 0.0) AS \"SALE_DISCOUNT\",
  i.item_id AS \"ITEM_ID\",
  i.item_discount AS \"ITEM_DISCOUNT\",
  i.weight AS \"WEIGHT\",
  si.quantity as \"STORE_QUANTITY\"
FROM stores s
INNER JOIN store_inventories si ON si.store_id = s.store_id
INNER JOIN items i ON si.item_id = i.item_id
LEFT OUTER JOIN department_discounts dd
  ON i.department_id = dd.department_id AND s.store_id = dd.store_id
WHERE s.store_id = $store_id AND i.item_id = (SELECT MIN(item_id)
  FROM store_inventories
  WHERE store_id = $store_id AND quantity > 0 AND item_id >
$my_item_id)
";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            if(!($rowref = $sth->fetchrow_hashref)) {
                  # no row returned... abort this transaction
                  return;
            }
            %row = %$rowref;
            $my_item_id = $row{ITEM_ID};
            $from_state = $row{FROM_STATE};
            $price = $row{PRICE};
            $store_discount = $row{STORE_DISCOUNT};
            $sale_discount = $row{SALE_DISCOUNT};
            $item_discount = $row{ITEM_DISCOUNT};
            $total_weight += $row{WEIGHT};
            $store_quantity = $row{STORE_QUANTITY};

            # load arrays for use later
            $item_id[$i] = $my_item_id;
            $item_price[$i] = $price;
            $discount[$i] = $store_discount + $sale_discount +
                  $item_discount;

            # apply discounts to come to final price, add to running
total
            $price = $price * (1 - $store_discount -
                  $item_discount - $sale_discount);
            $price = sprintf("%.2f", $price);
            $discounted_price[$i] = $price;
            $subtotal += $price;
            $sth->finish();

            # $dbh->do("LOCK TABLE store_inventories WRITE");
            if($store_quantity < 5) {
                  # automatically restock so the benchmark testing
                  # doesn't run out of inventory
                  $sql = "UPDATE store_inventories "
                  . "SET quantity = quantity + 9 "
                  . "WHERE store_id = $store_id and "
                  . "item_id = $my_item_id";
            } else {
                  # decrement store inventory quantity by 1 for this
item
                  $sql = "UPDATE store_inventories "
                  . "SET quantity = quantity - 1 "
                  . "WHERE store_id = $store_id and "
                  . "item_id = $my_item_id";
            }
            $dbh->do($sql);

            # $dbh->do("UNLOCK TABLES");
            $dbh->commit();         # avoid locks

      }

      # lookup volume discount
      $sql = "SELECT COALESCE(MAX(discount),0.00) AS
\"VOLUME_DISCOUNT\" "
            . "FROM volume_discounts WHERE total_purchase <
$total_price";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $rowref = $sth->fetchrow_hashref;
      %row = %$rowref;
      $volume_discount = $row{VOLUME_DISCOUNT};
      $sth->finish();
      $dbh->commit();         # avoid locks

      # apply volume and club discounts to subtotal
      $total_price = $subtotal * (1 - $volume_discount -
$club_discount);
      $total_price = sprintf("%.2f", $total_price);

      # lookup shipping
      $sql = "SELECT COALESCE(MIN(shipping_cost), 160.00) AS
\"SHIPPING\" "
            . "FROM shipping "
            . "WHERE from_state = '" . $from_state . "' AND to_state =
'"
            . $to_state . "' AND weight > $total_weight";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $rowref = $sth->fetchrow_hashref;
      %row = %$rowref;
      $shipping = $row{SHIPPING};
      $sth->finish();
      $dbh->commit();         # avoid locks
```

```
# apply shipping and tax to subtotal
$total_price += $shipping;
$total_price = sprintf("%.2f", $total_price * (1 + $tax/100.0));

# start tran - drops out of eval block on error
eval {

      # insert the transaction
      $sql = "INSERT INTO transactions VALUES($tran_seq "
            . "$customer_id, "
            . "$store_id, $now, $subtotal, $total_weight, "
            . "$club_discount, $volume_discount, "
            . "$shipping, $tax, $total_price)";
      $sql = fix_quotes($sql);
      $dbh->do("$sql");

      # get the insert id from the auto_increment
      if($dbtype eq "MySQL" or $dbtype eq "Sybase") {
            $transaction_id = $dbh->last_insert_id(undef,undef,
                  "transactions","transaction_id");
      } elsif($dbtype eq "Oracle") {
            $transaction_id =
                  get_oracle_sequence("transactions_seq");
      } elsif($dbtype eq "DB2") {
            $transaction_id = get_db2_last_insert_id();
      } elsif($dbtype eq "Firebird") {
            $transaction_id =
                  get_firebird_sequence("transactions_seq");
      } elsif($dbtype eq "Postgresql") {
            $transaction_id =
                  get_pg_sequence("transactions_seq");
      } else {
            die "Unknown dbtype: $dbtype";
      }


      # loop over items
      for($i=0; $i<$n_trans_items; $i++) {

            $sequence_no = $i + 1;
            # insert the transaction items
            $sql = "INSERT INTO transaction_items VALUES("
                  . "$transaction_id, $sequence_no, "
                  . "$item_id[$i], $item_price[$i], 1, "
                  . "$item_price[$i], "
                  . "$discount[$i], $discounted_price[$i])";
            $dbh->do("$sql");
      }

      # update club membership if applicable
      $sql = "UPDATE club_members SET total_quantity = "
            . "total_quantity + $n_trans_items, "
            . "total_spent = total_spent + $total_price, "
            . "last_purchase_date = $now "
            . "WHERE customer_id = $customer_id";
      $sql = fix_quotes($sql);
```

```
            $dbh->do("$sql");

            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # customer_purchases_item


sub adjust_inventory {

      # find the max item id
      $sql = "SELECT max(item_id) as \"ITEM_ID\" FROM items";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $rowref = $sth->fetchrow_hashref;
      %row = %$rowref;
      $max_item_id = $row{ITEM_ID};
      $sth->finish();
      $dbh->commit();
      $max_item_id = 25000 if($dbtype eq "Sybase");

      $item_id = int(rand()*($max_item_id-10));
      # find a valid item id, in case we picked a missing record
      $sql = "SELECT item_id as \"ITEM_ID\", "
            . "wholesale_price as \"WHOLESALE_PRICE\" "
            . "FROM items "
            . "WHERE item_id = (SELECT MIN(item_id) FROM items "
            . "WHERE item_id > $item_id)";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $rowref = $sth->fetchrow_hashref;
        %row = %$rowref;
      $item_id = $row{ITEM_ID};
      $wholesale_price = $row{WHOLESALE_PRICE};
      $sth->finish();
      $dbh->commit();

      # find the max store id
      $sql = "SELECT max(store_id) as \"STORE_ID\" FROM stores";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $rowref = $sth->fetchrow_hashref;
      %row = %$rowref;
      $max_store_id = $row{STORE_ID};
      $sth->finish();
      $dbh->commit();
      $max_store_id = 250 if($dbtype eq "Sybase");

      $store_id = int(rand()*($max_store_id-5));
      # find a valid store id, in case we picked a missing record
      $sql = "SELECT MIN(store_id) AS \"THIS_ID\" FROM stores "
            . "WHERE store_id > $store_id";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $rowref = $sth->fetchrow_hashref;
        %row = %$rowref;
      $store_id = $row{THIS_ID};
      $sth->finish();
      $dbh->commit();

      eval {
            $sql = "SELECT count(*) AS \"CNT\" FROM store_inventories "
                  . "WHERE store_id = $store_id AND "
                  . "item_id = $item_id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $sth->finish();
            if($row{CNT} == 0) {
                  # insert new inventory row
                  $quantity = int(rand()*6) + 1;
                    $price = sprintf("%.2f", $wholesale_price *
                        (1 + 0.05 * (int(rand()*8) + 1)));
                  $sql = "INSERT INTO store_inventories VALUES("
                  . "$store_id, $item_id, $quantity, $price)";
                  $dbh->do($sql);
            } else {
                  # delete inventory row
                  $sql = "DELETE FROM store_inventories WHERE "
                  . "store_id = $store_id AND item_id = $item_id";
                  $dbh->do($sql);
            }
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # adjust_inventory


sub adjust_item_for_all_stores {

      # find the max item id
      $sql = "SELECT max(item_id) as \"ITEM_ID\" FROM items";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $rowref = $sth->fetchrow_hashref;
      %row = %$rowref;
      $max_item_id = $row{ITEM_ID};
      $sth->finish();
      $dbh->commit();
      $max_item_id = 25000 if($dbtype eq "Sybase");

      # Pick an item at random
      $item_id = int(rand()*($max_item_id-10));
      $sql = "SELECT count(*) AS \"CNT\" FROM items WHERE item_id =
$item_id";
```

```
$sth = $dbh->prepare($sql);
$sth->execute();
$rowref = $sth->fetchrow_hashref;
      %row = %$rowref;
$sth->finish();
$dbh->commit();

if($row{CNT} == 0) {
      # add an item if it does not exist
      $dbh->commit();         # We're done with thie transaction.
      create_new_item();      # Runs its own transaction.
      return;
}
$dbh->commit();         # prevent locks

# We need the store ids
undef @these_stores;
$sql = "SELECT store_id as \"STORE_ID\" FROM stores";
$sth = $dbh->prepare($sql);
$sth->execute();
while($rowref = $sth->fetchrow_hashref) {
            %row = %$rowref;
      push(@these_stores, $row{STORE_ID});
}
$sth->finish();
$dbh->commit();         # avoid locks

# to keep the db from exhausting supplies, we sometimes
# add quantity instead of remove quantity.
if(rand() < 0.3) {
      # set the store inventories to zero quantity
      $sql = "UPDATE store_inventories SET quantity = 0 "
            . "WHERE store_id = ? AND item_id = ?";
} else {
      # add 20 items to each store that carries it
      $sql = "UPDATE store_inventories "
            . "SET quantity = quantity + 20 "
            . "WHERE store_id = ? AND item_id = ?";
}
$sth = $dbh->prepare($sql);

eval {
      # $dbh->do("LOCK TABLE store_inventories WRITE");
      foreach $store_id (@these_stores) {
            # Loop over all stores.  Where stores do not
            # exist in store_inventories, nothing will
            # happen and that's okay.
            #
            $sth->execute($store_id, $item_id);
      }
      # $dbh->do("UNLOCK TABLES");
      $dbh->commit();
};
if($@) {
      warn "Database error: $DBI::errstr\n";
      $dbh->rollback(); # dies if rollback fails
}
```

```
} # adjust_item_for_all_stores


sub update_phone {

      # select a customer account randomly
      $id = int(rand()*($INITIAL_CUSTOMERS-10));

      # current timestamp
      ($sec,$min,$hr,undef,undef,undef) = localtime(time);
      # Put timestamp in 1/1/08
      $now = timelocal($sec,$min,$hr,1,0,2008);
      # Convert to general format
      $now = '"' . my_format($now) . '"';
      # Convert to DB-specific format
      $now = fix_one_date($now);

      # start tran - drops out of eval block on error
      eval {
            # find a valid id, in case we picked a missing record
            $sql = "SELECT MIN(customer_id) AS \"THIS_ID\" "
                  . "FROM customer_accounts "
                  . "WHERE customer_id > $id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $customer_id = $row{THIS_ID};
            $sth->finish();

            $new_phone = generate_phone();

            # update the phone number
            $sql = "UPDATE customer_accounts SET phone = $new_phone, "
                  . "activity_date = $now "
                  . "WHERE customer_id = $customer_id";
            $sql = fix_quotes($sql);
            $dbh->do($sql);

            # commit tran
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # update_phone


sub update_address {

      # select a customer account randomly
      $id = int(rand()*($INITIAL_CUSTOMERS-10));

      # current timestamp
      ($sec,$min,$hr,undef,undef,undef) = localtime(time);
      # Put timestamp in 1/1/08
      $now = timelocal($sec,$min,$hr,1,0,2008);
      # Convert to general format
      $now = '"' . my_format($now) . '"';
      # Convert to DB-specific format
      $now = fix_one_date($now);

      # start tran - drops out of eval block on error
      eval {
            # find a valid id, in case we picked a missing record
            $sql = "SELECT MIN(customer_id) AS \"THIS_ID\" "
                  . "FROM customer_accounts "
                  . "WHERE customer_id > $id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $customer_id = $row{THIS_ID};
            $sth->finish();

            # find a valid sequence number, in case we picked
            # a missing record
            $sql = "SELECT MIN(sequence_number) AS \"SEQ_NO\" "
                  . "FROM customer_addresses "
                  . "WHERE customer_id = $customer_id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $seq_no = $row{SEQ_NO};

            $new_address = generate_address();
            $city_state_zip = get_random_city();
            ($new_city, $new_state, $new_zip) = split /,/,
$city_state_zip;
            $sth->finish();

            # update the activity date
            $sql = "UPDATE customer_accounts SET activity_date = $now "
                  . "WHERE customer_id = $customer_id";
            $sql = fix_quotes($sql);
            $dbh->do($sql);

            # update the address
            $sql = "UPDATE customer_addresses "
                  . "SET street_address = $new_address, "
                  . "city = $new_city, state = $new_state, "
                  . "zip = $new_zip "
                  . "WHERE customer_id = $customer_id "
                  . "AND sequence_number = $seq_no";
            $sql = fix_quotes($sql);
            $dbh->do($sql);

            # commit tran
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # update_address


sub add_new_address {

      # select a customer account randomly
      $id = int(rand()*($INITIAL_CUSTOMERS-10));

      # current timestamp
      ($sec,$min,$hr,undef,undef,undef) = localtime(time);
      # Put timestamp in 1/1/08
      $now = timelocal($sec,$min,$hr,1,0,2008);
      # Convert to general format
      $now = '"' . my_format($now) . '"';
      # Convert to DB-specific format
      $now = fix_one_date($now);

      # start tran - drops out of eval block on error
      eval {
            # find a valid id, in case we picked a missing record
            $sql = "SELECT MIN(customer_id) AS \"THIS_ID\" "
                  . "FROM customer_accounts "
                  . "WHERE customer_id > $id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $customer_id = $row{THIS_ID};
            $sth->finish();

            # find a valid id, in case we picked a missing record
            $sql = "SELECT MAX(sequence_number) AS \"MAX_SEQ_NO\" "
                  . "FROM customer_addresses "
                  . "WHERE customer_id = $customer_id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $sequence_number = $row{MAX_SEQ_NO} + 1;

            $new_address = generate_address();
            $city_state_zip = get_random_city();
            ($new_city, $new_state, $new_zip) = split /,/,
$city_state_zip;
            $sth->finish();

            # update the activity date
            $sql = "UPDATE customer_accounts SET activity_date = $now "
                  . "WHERE customer_id = $customer_id";
            $sql = fix_quotes($sql);
            $dbh->do($sql);
            # insert the new address
            $sql = "INSERT INTO customer_addresses VALUES($customer_id,
"
                  . "$sequence_number, $new_address, $new_city, "
                  . "$new_state, $new_zip)";
            $sql = fix_quotes($sql);
            $dbh->do($sql);

            # commit tran
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # add_new_address


sub change_current_address {

      # select a customer account randomly
      $id = int(rand()*($INITIAL_CUSTOMERS-10));

      # current timestamp
      ($sec,$min,$hr,undef,undef,undef) = localtime(time);
      # Put timestamp in 1/1/08
      $now = timelocal($sec,$min,$hr,1,0,2008);
      # Convert to general format
      $now = '"' . my_format($now) . '"';
      # Convert to DB-specific format
      $now = fix_one_date($now);

      # start tran - drops out of eval block on error
      eval {
            # find a valid id, in case we picked a missing record
            $sql = "SELECT MIN(customer_id) AS \"THIS_ID\" "
                  . "FROM customer_accounts "
                  . "WHERE customer_id > $id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $customer_id = $row{THIS_ID};
            $sth->finish();

            # find a valid sequence number, in case we picked
            # a missing record
            $sql = "SELECT MAX(sequence_number) AS \"SEQ_NO\" "
                  . "FROM customer_addresses "
                  . "WHERE customer_id = $customer_id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $new_seq_no = int(rand()*$row{SEQ_NO}) + 1;
            $sth->finish();
            # find a valid sequence number, in case we picked
            # a missing record
            $sql = "SELECT MIN(sequence_number) AS \"SEQ_NO\" "
                  . "FROM customer_addresses "
                  . "WHERE customer_id = $customer_id AND "
                  . "sequence_number >= $new_seq_no";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $new_address_number = $row{SEQ_NO};
            $sth->finish();

            # update the activity date and new current address
            $sql = "UPDATE customer_accounts SET activity_date = $now,
"
                  . "current_address = $new_address_number "
                  . "WHERE customer_id = $customer_id";
            $sql = fix_quotes($sql);
            $dbh->do($sql);

            # commit tran
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }


} # change_current_address


sub customer_joins_club {

      # select a customer account randomly
      $id = int(rand()*($INITIAL_CUSTOMERS-10));

      # current timestamp
      ($sec,$min,$hr,undef,undef,undef) = localtime(time);
      # Put timestamp in 1/1/08
      $now = timelocal($sec,$min,$hr,1,0,2008);
      # Convert to general format
      $now = '"' . my_format($now) . '"';
      # Convert to DB-specific format
      $now = fix_one_date($now);

      # start tran - drops out of eval block on error
      eval {
            # find a valid id, in case we picked a missing record
            $sql = "SELECT MIN(customer_id) AS \"THIS_ID\" "
                  . "FROM customer_accounts "
                  . "WHERE customer_id > $id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
```

```
        $rowref = $sth->fetchrow_hashref;
        %row = %$rowref;
        $customer_id = $row{THIS_ID};
        $sth->finish();

        $sql = "SELECT COALESCE(SUM(discount),0) AS \"DISCOUNT\" "
            . "FROM club_members "
            . "WHERE customer_id = $customer_id";
        $sth = $dbh->prepare($sql);
        $sth->execute();
        $rowref = $sth->fetchrow_hashref;
        %row = %$rowref;
        $sth->finish();

        if($row{DISCOUNT} == 0) {
            # insert new row to club members table
            $sql = "INSERT INTO club_members "
                . "VALUES($customer_id, 0, 0.0, "
                . "$now, null, 0.05)";
        } elsif($row{DISCOUNT} > 0.14) {
            # delete this membership
            $sql = "DELETE FROM club_members "
                . "WHERE customer_id = $customer_id";
        } else {
            # increase this membership level
            $sql = "UPDATE club_members "
                . "SET discount = discount + 0.05 "
                . "WHERE customer_id = $customer_id";
        }
        $sql = fix_quotes($sql);
        $dbh->do($sql);

        # update the activity date
        $sql = "UPDATE customer_accounts SET activity_date = $now "
            . "WHERE customer_id = $customer_id";
        $sql = fix_quotes($sql);
        $dbh->do($sql);

        # commit tran
        $dbh->commit();
    };
    if($@) {
        warn "Database error: $DBI::errstr\n";
        $dbh->rollback(); # dies if rollback fails
    }

} # customer_joins_club


sub create_new_item {

    # generate item characteristics
    @adjective = ("Fast", "Slow", "Premium", "Cool", "Hot",
"Stylish",
        "Sophisticated", "Gentle", "Rough");
    @noun = ("Sweater", "Coat", "Pants", "Shoes", "Socks", "Jacket",
        "Suit", "Sofa", "Chair");
```

```
      @color = ("Red", "Blue", "Yellow", "Green", "Purple", "Orange",
            "Pink", "Black", "White", "Gray");
      @size = ("Small", "Medium", "Large", "Petite", "Big", "Tall",
"S",
            "XS", "M", "L", "XL", "Junior");
      @brand = ("Smyth", "Jonez", "Lightyear", "Ralf", "GKline",
"Gooch");
      @style = ("Modern", "Spring", "Fall", "Summer", "Winter",
"Casual",
            "Business");
      @cloth = ("Felt", "Cotton", "Silk", "Nylon", "Rayon", "Leather",
            "Corduroy", "Chenille", "Elastic", "Spandex");

      $words[0] = \@adjective;       $n_words[0] = $#adjective;
      $words[1] = \@noun;      $n_words[1] = $#noun;
      $words[2] = \@color;     $n_words[2] = $#color;
      $words[3] = \@size;      $n_words[3] = $#size;
      $words[4] = \@brand;     $n_words[4] = $#brand;
      $words[5] = \@style;     $n_words[5] = $#style;
      $words[6] = \@cloth;     $n_words[6] = $#cloth;

      $words = int(rand()*4) + 2;
      $name = "";
      for ($w=0;$w<$words;$w++) {
            $wordtype = int(rand()*7);
            $word_int = int(rand()*$n_words[$wordtype]);
            $name .= "$words[$wordtype][$word_int] ";
      }
      chop($name);
      if(length($name) > 40) {
            $name = substr($name, 0, 40);
      }
      $name = "\"$name\"";

      $price = sprintf("%.2f", 10.0 + (rand()*6  + 1) * (rand()*100));
      $weight = 1 + int(rand()*20);
      $d = rand();
      if($d < 0.2) {
            $discount = 0.05 * (int($d * 5 * 4) + 1);
      } else {
            $discount = 0.00;
      }
      $department = int(rand()*$INITIAL_DEPARTMENTS) + 1;

      eval {
            $sql = "INSERT INTO items VALUES($items_seq $name, $price,
"
                  . "$weight, $discount, $department)";
            $sql = fix_quotes($sql);
            $dbh->do($sql);

            # get the insert id from the auto_increment
            if($dbtype eq "MySQL" or $dbtype eq "Sybase") {
                  $item_id = $dbh->last_insert_id(undef,undef,
                        "items","item_id");
            } elsif($dbtype eq "Oracle") {
                  $item_id =
                  get_oracle_sequence("items_seq");
      } elsif($dbtype eq "DB2") {
            $item_id = get_db2_last_insert_id();
      } elsif($dbtype eq "Postgresql") {
            $item_id =
                  get_pg_sequence("items_seq");
      } elsif($dbtype eq "Firebird") {
            $item_id =
                  get_firebird_sequence("items_seq");
      } else {
            die "Unknown dbtype: $dbtype";
      }
      $dbh->commit();         # avoid locks
};
if($@) {
      warn "Database error: $DBI::errstr\n";
      $dbh->rollback(); # dies if rollback fails
      return;
}

# 20% chance to add item to store inventories
if(rand() > 0.2) {
      return;
}

# start tran
eval {

      # We need the store ids
      undef(@these_stores);
      $sql = "SELECT store_id as \"STORE_ID\" FROM stores ";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      while($rowref = $sth->fetchrow_hashref) {
                  %row = %$rowref;
            push(@these_stores, $row{STORE_ID});
      }
      $dbh->commit();         # avoid locks

      foreach $store_id (@these_stores) {

            # 40% chance this store gets no inventory
            next if(rand() < 0.4);

            # otherwise 1-6 quantity
            $qty = int(rand()*6) + 1;

                  # retail = wholesale plus 5-40% markup
                  $markup = 0.05 * (int(rand()*8) + 1);
            $retail_price = sprintf("%.2f",
                        $price * (1 + $markup));

            $sql = "INSERT INTO store_inventories VALUES("
                  . "$store_id, $item_id, "
                  . "$qty, $retail_price)";
            $dbh->do($sql);
            $dbh->commit();         # avoid locks
            }
            # don't forget to empty out this array, we will
            # be re-using it later
            undef(@these_stores);

            # commit tran
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # create_new_item


sub update_item {

      # Select random item
      $item_id = int(rand()*($INITIAL_ITEMS-10));

      # find a valid item id, in case we picked a missing record
      $sql = "SELECT item_id as \"ITEM_ID\", "
            . "wholesale_price as \"WHOLESALE_PRICE\" "
            . "FROM items "
            . "WHERE item_id = (SELECT MIN(item_id) FROM items "
            . "WHERE item_id > $item_id)";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $rowref = $sth->fetchrow_hashref;
        %row = %$rowref;
      $item_id = $row{ITEM_ID};
      $old_price = $row{WHOLESALE_PRICE};
      $sth->finish();
      $dbh->commit();

      # Randomly generate new price and discount
      $new_price = sprintf("%.2f", 10.0 + (rand()*6  + 1) *
(rand()*100));
      $d = rand();
      if($d < 0.2) {
            $discount = 0.05 * (int($d * 5 * 4) + 1);
      } else {
            $discount = 0.00;
      }

      # Compute markup
      $markup = sprintf("%.2f", $new_price - $old_price);

      # start tran - drops out of eval block on error
      eval {

            # Update item price and discount
            $sql = "UPDATE items SET wholesale_price = $new_price, "
                  . "item_discount = $discount WHERE item_id =
$item_id";
            $dbh->do($sql);
            # Apply markup to store inventories for that item
            $sql = "UPDATE store_inventories SET retail_price = "
                  . "retail_price + ($markup) WHERE item_id =
$item_id";
            $dbh->do($sql);

            # commit tran
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # update_item


sub create_new_store {

      # generate data for new store
      @a = ("Mega", "Super", "Mini", "Shop");
      @b = ("Mart", "Online", "Catalog");
      @fifty_states = ("AK","AL","AR","AZ","CA","CO","CT","DE","FL",
            "GA","HI","IA","ID","IL","IN","KS","KY","LA","MA","MD",
            "ME","MI","MN","MO","MS","MT","NC","ND","NE","NH","NJ",
            "NM","NV","NY","OH","OK","OR","PA","RI","SC","SD","TN",
            "TX","UT","VA","VT","WA","WI","WV","WY");
      $a_int = int(rand()*4);
      $b_int = int(rand()*3);
      $st_int = int(rand()*50);

      $d = rand();
      if($d < 0.8) {
            $discount = "0.00";
      } elsif($d < 0.85) {
            $discount = "0.05";
      } elsif($d < 0.90) {
            $discount = "0.10";
      } elsif($d < 0.95) {
            $discount = "0.15";
      } else {
            $discount = "0.20";
      }

      $name = "$a[$a_int] $b[$b_int]";
      $from_state = "\"$fifty_states[$st_int]\"";

      # start tran - drops out of eval block on error
      eval {
            # Get store number
            $sql = "SELECT MAX(store_id) AS \"STORE_ID\" FROM stores";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $name = sprintf("\"$name #%d\"", $row{STORE_ID} + 1);
```

```
$sth->finish();

# insert into stores table
$sql = "INSERT INTO stores VALUES($stores_seq $name, "
      . "$discount, $from_state)";
$sql = fix_quotes($sql);
$dbh->do($sql);

# Get the created store id
if($dbtype eq "MySQL" or $dbtype eq "Sybase") {
      $store_id = $dbh->last_insert_id(undef,undef,
            "stores","store_id");
} elsif($dbtype eq "Oracle") {
      $store_id =
            get_oracle_sequence("stores_seq");
} elsif($dbtype eq "DB2") {
      $store_id = get_db2_last_insert_id();
} elsif($dbtype eq "Postgresql") {
      $store_id =
            get_pg_sequence("stores_seq");
} elsif($dbtype eq "Firebird") {
      $store_id =
            get_firebird_sequence("stores_seq");
} else {
      die "Unknown dbtype: $dbtype";
}
$dbh->commit();          # avoid locks

# generate data for department discounts
# for each of the 60 departments
for($i=1; $i<=$INITIAL_DEPARTMENTS; $i++) {

      # 20% chance this store has this dept
      next if(rand() > 0.2);

      $d = rand();
      if($d < 0.5) {
            $discount = "0.00";
      } elsif($d < 0.60) {
            $discount = "0.05";
      } elsif($d < 0.70) {
            $discount = "0.10";
      } elsif($d < 0.80) {
            $discount = "0.20";
      } elsif($d < 0.90) {
            $discount = "0.30";
      } else {
            $discount = "0.40";
      }

      $sql = "INSERT INTO department_discounts "
            . "VALUES($store_id, $i, $discount)";
      $dbh->do($sql);

} # for each department
$dbh->commit;            # avoid locks
```

```
            # generate data for new store inventories
            undef %price_hash;
            $sql = "SELECT item_id as \"ITEM_ID\", "
                  . "wholesale_price as \"WHOLESALE_PRICE\" FROM
items";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            while($rowref = $sth->fetchrow_hashref) {
                  %row = %$rowref;
                  $item_id = $row{ITEM_ID};
                  $price_hash{$item_id} = $row{WHOLESALE_PRICE};
            }
            $dbh->commit;           # avoid locks

            foreach $item_id (keys %price_hash) {
                  # 99% chance skip, else 1-6 qty.
                  next if(rand() < 0.99);
                  $qty = int(rand()*6) + 1;

                  # retail = wholesale plus 5-40% markup
                  $markup = 0.05 * (int(rand()*8) + 1);
                  $retail_price = sprintf("%.2f",
                        $price_hash{$item_id} * (1 + $markup));

                  # insert this inventory
                  $sql = "INSERT INTO store_inventories "
                        . "VALUES($store_id, $item_id, $qty, "
                        . "$retail_price)";
                  $dbh->do($sql);
                  $dbh->commit;           # avoid locks

            } # for each item, going into store inventory

            # clear this hash when done
            undef %price_hash;

            # commit tran
            $dbh->commit();
      };

      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # create_new_store


sub close_store {

      # start tran - drops out of eval block on error
      eval {
            # find the max store id
            $sql = "SELECT max(store_id) as \"STORE_ID\" FROM stores";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $max_id = $row{STORE_ID};
            $sth->finish();
            $max_id = 250 if($dbtype eq "Sybase");

            $store_id = int(rand()*($max_id-5));
            # check to see if store_id exists
            $sql = "SELECT count(*) AS \"CNT\" FROM stores "
                  . "WHERE store_id = $store_id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $sth->finish();

            if($row{CNT} == 0) {

                  # No store found-- create new store instead.
                  create_new_store();

            } else {

                  # Delete this store and records related to it.
                  $sql = "DELETE FROM store_inventories "
                        . "WHERE store_id = $store_id";
                  $dbh->do("$sql");

                  $sql = "DELETE FROM department_discounts "
                        . "WHERE store_id = $store_id";
                  $dbh->do("$sql");

                  $sql = "DELETE FROM stores "
                        . "WHERE store_id = $store_id";
                  $dbh->do("$sql");

            }

            # commit tran
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # close_store

sub update_store_discount {

      $store_id = int(rand()*($INITIAL_STORES-5));

      # start tran - drops out of eval block on error
      eval {
            # find a valid store id, in case we picked a missing record
            $sql = "SELECT store_id as \"STORE_ID\", "
                  . "store_discount as \"STORE_DISCOUNT\" FROM stores "
                  . "WHERE store_id = (SELECT MIN(store_id) FROM stores
"
                  . "WHERE store_id > $store_id)";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $store_id = $row{STORE_ID};
            $old_discount = $row{STORE_DISCOUNT};
            $sth->finish();

            if($old_discount < 0.2) {
                  $new_discount = sprintf("%.2f", $old_discount +
0.05);
                  $factor = 1.05;
            } else {
                  $new_discount = 0.0;
                  $factor = 0.95;
            }
            $dbh->commit();   # avoid locks

            # update the store's discount
            $sql = "UPDATE stores SET store_discount = $new_discount "
                  . "WHERE store_id = $store_id";
            $dbh->do($sql);
            $dbh->commit();   # avoid locks

            # update the retail prices in the store's inventory
            $sql = "UPDATE store_inventories "
                  . "SET retail_price = retail_price * $factor "
                  . "WHERE store_id = $store_id";
            $dbh->do($sql);

            # commit tran
            $dbh->commit();
      };

      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # update_store_discount


sub adjust_department_discount {

      $store_id = int(rand()*($INITIAL_STORES-10));
      $department_id = int(rand()*$INITIAL_DEPARTMENTS) + 1;

      # start tran - drops out of eval block on error
      eval {
            # find a valid store id, in case we picked a missing record
            $sql = "SELECT MIN(store_id) AS \"STORE_ID\" FROM stores "
                  . "WHERE store_id > $store_id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $store_id = $row{STORE_ID};
            $sth->finish();

            $sql = "SELECT count(*) AS \"CNT\" FROM
department_discounts "
                  . "WHERE store_id = $store_id AND "
                  . "department_id = $department_id";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $sth->finish();

            if($row{CNT} == 0) {
                  # create a new record
                  $d = rand();
                  if($d < 0.5) {
                        $discount = "0.00";
                  } elsif($d < 0.60) {
                        $discount = "0.05";
                  } elsif($d < 0.70) {
                        $discount = "0.10";
                  } elsif($d < 0.80) {
                        $discount = "0.20";
                  } elsif($d < 0.90) {
                        $discount = "0.30";
                  } else {
                        $discount = "0.40";
                  }

                  $sql = "INSERT INTO department_discounts "
                        . "VALUES($store_id, $department_id, "
                        . "$discount)";
                  $dbh->do($sql);
            } else {
                  # delete the record
                  $sql = "DELETE FROM department_discounts "
                        . "WHERE store_id = $store_id AND "
                        . "department_id = $department_id";
                  $dbh->do($sql);
            }

            # commit tran
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            $dbh->rollback(); # dies if rollback fails
      }

} # adjust_department_discount


sub update_volume_discount {
   # start tran - drops out of eval block on error
   eval {

      if(rand() < 0.5) {
         $factor = 0.9;
      } else {
         $factor = 1.1;
      }

      # Need this to avoid deadlocks
      if($dbtype eq "Sybase") {
         $dbh->do("BEGIN TRAN");
      }
      if($dbtype ne "Firebird") {
         $dbh->do("LOCK TABLE volume_discounts
$tablelockmode");
      }

        # see if this item's already been rating by this user
        if($dbtype eq "Firebird") {
            # Firebird's locking is nonstandard
            $sql  = "SELECT * FROM volume_discounts with
lock";

            $sth = $dbh->prepare($sql);
            $sth->execute();
            $sth->finish();
        }

      # Regenerate the table from scratch to keep the values
      # reasonable.
      $dbh->do("DELETE FROM volume_discounts");

      $value = 100 * $factor;
      $dbh->do("INSERT INTO volume_discounts
VALUES($value,0.00)");
      for($i=1;$i<=10;$i++) {
         $value = 2 * $value;
         $sql = sprintf("INSERT INTO volume_discounts "
            . "VALUES(%d,%.2f)", $value, 0.01 * $i);
         $dbh->do($sql);
      }

      # commit tran
      if($dbtype eq "Sybase") {
         $dbh->do("COMMIT TRAN");
      }
      if($dbtype eq "MySQL") {
         $dbh->do("UNLOCK TABLES");
      }
      $dbh->commit();
   };
   if($@) {
      warn "Database error: $DBI::errstr\n";
      if($dbtype eq "Sybase") {
         $dbh->do("ROLLBACK TRAN");
      }
      if($dbtype eq "MySQL") {
```

```
                  $dbh->do("UNLOCK TABLES");
            }
            $dbh->rollback(); # dies if rollback fails
      }

} # update_volume_discount


sub update_shipping {

      $sql_increase = "UPDATE shipping "
            . "SET shipping_cost = shipping_cost * 1.1";
      $sql_decrease = "UPDATE shipping "
            . "SET shipping_cost = shipping_cost / 1.1";

      # start tran - drops out of eval block on error
      eval {
            if($dbtype eq "Sybase") {
                  $dbh->do("BEGIN TRAN");
            }
            if($dbtype ne "Firebird") {
                  $dbh->do("LOCK TABLE shipping $tablelockmode");
            }

                # see if this item's already been rating by this user
                if($dbtype eq "Firebird") {
                        # Firebird's locking is nonstandard
                        $sql  = "SELECT * FROM shipping with lock";
                        $sth = $dbh->prepare($sql);
                        $sth->execute();
                        $sth->finish();
                }

            # Get the minimum shipping
            $sql = "SELECT MIN(shipping_cost) AS \"THE_MIN\" FROM
shipping";
            $sth = $dbh->prepare($sql);
            $sth->execute();
            $rowref = $sth->fetchrow_hashref;
            %row = %$rowref;
            $sth->finish();

            if($row{THE_MIN} < 4.95) {
                  # if it's less than X, increase 10%
                  $sql = $sql_increase;
            } elsif($row{THE_MIN} > 13.95) {
                  # if it's greater than Y, decrease 10%
                  $sql = $sql_decrease;
            } elsif(rand() < 0.5) {
                  # else 50/50 chance
                  $sql = $sql_increase;
            } else {
                  $sql = $sql_decrease;
            }

            # Update shipping
            $dbh->do($sql);
            # commit tran
            if($dbtype eq "Sybase") {
                  $dbh->do("COMMIT TRAN");
            }
            if($dbtype eq "MySQL") {
                  $dbh->do("UNLOCK TABLES");
            }
            $dbh->commit();
      };
      if($@) {
            warn "Database error: $DBI::errstr\n";
            if($dbtype eq "Sybase") {
                  $dbh->do("ROLLBACK TRAN");
            }
            if($dbtype eq "MySQL") {
                  $dbh->do("UNLOCK TABLES");
            }
            $dbh->rollback(); # dies if rollback fails
      }

} # update_shipping


#sub update_club_member_discount() {
#
#     $sql_increase = "UPDATE club_members SET discount = discount *
1.1";
#     $sql_decrease = "UPDATE club_members SET discount = discount /
1.1";
#
#     # Get the minimum shipping
#     $sql = "SELECT MAX(discount) AS the_max FROM club_members";
#     $sth = $dbh->prepare($sql);
#     $sth->execute();
#     $rowref = $sth->fetchrow_hashref;
#     %row = %$rowref;
#
#     if($row{the_max} < 0.12) {
#           # if it's less than X, increase 10%
#           $sql = $sql_increase;
#     } elsif($row{the_max} > 0.21) {
#           # if it's greater than Y, decrease 10%
#           $sql = $sql_decrease;
#     } elsif(rand() < 0.5) {
#           # else 50/50 chance
#           $sql = $sql_increase;
#     } else {
#           $sql = $sql_decrease;
#     }
#     $dbh->commit();         # reduce locks
#
#     # start tran - drops out of eval block on error
#     eval {
#           # Update shipping
#           $dbh->do($sql);
#
```

```
#           # commit tran
#           $dbh->commit();
#     };
#
#     if($@) {
#           warn "Database error: $DBI::errstr\n";
#           $dbh->rollback(); # dies if rollback fails
#     }
#
#} # update_club_member_discount


#
# utility functions
#

sub get_random_name {

      if(rand() < 0.52) {
            $first_index = rand() * 43.085;
            $first_name = select_name($first_index, \@female_names,
                  \@female_values);
      } else {
            $first_index = rand() * 59.531;
            $first_name = select_name($first_index, \@male_names,
                  \@male_values);
      }
      $last_index = rand() * 18.825;

      $last_name = select_name($last_index, \@last_names,
\@last_values);

      return "\"$first_name\",\"$last_name\"";

} # get_random_name


sub select_name {

      my($index, $a, $b) = @_;

      @names = @{$a};
      @values = @{$b};

      for(my $i = 0; $i < $#names + 1; $i++) {

            if($index < $values[$i]) {
                  return $names[$i];
            }
      }
      print "Failed to find name!!\n";

} # select_name


sub get_random_city {
      $zip = sprintf "%05d", int (rand() * 100000);

      $index = rand() * 0.400486605;

      for(my $i = 0; $i < $#city_names + 1; $i++) {

            if($index < $city_values[$i]) {
                  return "\"$city_names[$i]\",\"$states[$i]\",$zip";
            }
      }
      print "Failed to find a city!  How is that possible?!\n";

} # get_random_city


sub generate_phone {

      return sprintf("\"%03d-%03d-%04d\"", int (rand() * 700) + 200,
            int(rand()*1000), int(rand()*10000));

} # generate_phone


sub generate_address {

      $house = int (rand() * 10000) + 1;
      $street = int (rand() * 150) + 1;
      if($street % 10 == 1) {
            $street .= "st";
      } elsif($street % 10 == 2) {
            $street .= "nd";
      } elsif($street % 10 == 3) {
            $street .= "rd";
      } else {
            $street .= "th";
      }

      $d = int(rand() * 20)+1;
      if($d > 18) {
            $dir = "East ";
      } elsif($d > 16) {
            $dir = "West ";
      } elsif($d > 14) {
            $dir = "South ";
      } elsif($d > 12) {
            $dir = "North ";
      } elsif($d > 11) {
            $dir = "NE ";
      } elsif($d > 10) {
            $dir = "NW ";
      } elsif($d > 9) {
            $dir = "SE ";
      } elsif($d > 8) {
            $dir = "SW ";
      } else {
            $dir = "";
      }
      $s = int(rand() * 4);
      $str = "Road" if $s == 0;
      $str = "Street" if $s == 1;
      $str = "Avenue" if $s == 2;
      $str = "Boulevard" if $s == 3;

      return "\"$house $dir$street $str\"";

} # generate_address


sub my_format() {

      $d = $_[0];

      ($sec,$min,$hr,$day,$mon,$yr) = localtime($d);

      return sprintf("%02d/%02d/%4d %02d:%02d:%02d", $mon+1, $day,
$yr+1900,
            $hr, $min, $sec);

} # my_format


sub load_data {

      $INITIAL_CUSTOMERS = 500000;
      $INITIAL_ITEMS = 25000;
      $INITIAL_STORES = 250;
      $INITIAL_DEPARTMENTS = 60;

      my $i = 0;
      open FILE, "<gen/100male.txt";
      while (<FILE>) {
            chop;
            ($male_names[$i], $male_values[$i]) = split /,/;
            $i++;
      }
      close FILE;

      $i = 0;
      open FILE, "<gen/100female.txt";
      while (<FILE>) {
            chop;
            ($female_names[$i], $female_values[$i]) = split /,/;
            $i++;
      }
      close FILE;

      $i = 0;
      open FILE, "<gen/100lastnames.txt";
      while (<FILE>) {
            chop;
            ($last_names[$i], $last_values[$i]) = split /,/;
            $i++;
      }
```

```
        close FILE;

        $i = 0;
        open FILE, "<gen/100cities.txt";
        while (<FILE>) {
                chop;
                ($city_values[$i], $states[$i], $ci) = split /,/;
                # Need to truncate the city names to 20 characters.
                $city_names[$i] = substr($ci,0,20);
                $i++;
        }
        close FILE;

} # load_data


sub get_oracle_sequence {

        $sequence = shift;

        $sql = "select $sequence.currval as SEQ from dual";
        $sth = $dbh->prepare($sql);
        $sth->execute();
        $rowref = $sth->fetchrow_hashref();
        return $rowref->{'SEQ'};

} # get_oracle_sequence


sub get_pg_sequence {

        $sequence = shift;

        $sql = "select currval('$sequence') as \"SEQ\"";
        $sth = $dbh->prepare($sql);
        $sth->execute();
        $rowref = $sth->fetchrow_hashref();
        return $rowref->{'SEQ'};

} # get_pg_sequence


sub get_firebird_sequence {

        $sequence = shift;

        $sql = "select gen_id($sequence, 0) as GENID from rdb\$database";
        $sth = $dbh->prepare($sql);
        $sth->execute();
        $rowref = $sth->fetchrow_hashref();
        return $rowref->{'GENID'};

} # get_firebird_sequence


sub get_db2_last_insert_id {
	$sql = "select identity_val_local() as LASTVAL from
sysibm.sysdummy1";
	$sth = $dbh->prepare($sql);
	$sth->execute();
	$ref = $sth->fetchrow_hashref();
	return $ref->{'LASTVAL'};

} # get_db2_last_insert_id


sub db2_convert_date {
	$old_date = shift;
	return "null" if $old_date eq '';

	$old_date =~ /"(.*)"/;		# strip quotes
	$old_date = $1;
	my ($mon, $day, $yr, $hr, $min, $sec);
	my @p = split / /, $old_date;
	($mon, $day, $yr) = split /\//, $p[0];
	($hr, $min, $sec) = split /:/, $p[1];
	return sprintf("\"%d-%02d-%02d %02d:%02d:%02d\"", $yr, $mon,
$day,
		$hr, $min, $sec);

} # db2_convert_date


# Change the quotes and escapes if necessary.
#
sub fix_quotes {
	my $str = shift;

	switch($dbtype) {
		case "MySQL" { } # No changes needed.
		case "Oracle" { # Change ' to '' and " to '.
			$str =~ s/'/''/g;
			$str =~ s/"/'/g;
		}
		case "DB2" { # Change ' to '' and " to '.
			$str =~ s/'/''/g;
			$str =~ s/"/'/g;
		}
		case "Postgresql" { # Change ' to '' and " to '.
			$str =~ s/'/''/g;
			$str =~ s/"/'/g;
		}
		case "Sybase" { # Change ' to '' and " to '.
			$str =~ s/'/''/g;
			$str =~ s/"/'/g;
		}
		case "Firebird" { # Change ' to '' and " to '.
			$str =~ s/'/''/g;
			$str =~ s/"/'/g;
		}
		default { die "DBD $dbtype not found."; }
	}
	return $str;
```

```
} # fix_quotes
```

Appendix O:

Benchmark Script for Report Generation

```
#!/usr/bin/perl -w
#
# reportb.pl - run report queries in parallel as part of the benchmark
case
#

$dbtype = "DB2";
#$dbtype = "MySQL";
#$dbtype = "Firebird";
#$dbtype = "Oracle";
#$dbtype = "Postgresql";
#$dbtype = "Sybase";

use Time::HiRes qw(gettimeofday tv_interval);    # High resolution
timing
use Time::Local;                    # reverse of localtime
use Switch;                         # For switch/case statements
use DBI;                            # General database interface
use DBD::DB2;                 # DB2 specific interface
use DBD::DB2::Constants;      # more for DB2
#use DBD::mysql;        # MySQL specific interface
#use DBD::InterBase;          # Firebird specific interface
#use DBD::Oracle;       # Oracle specific interface
#use DBD::Pg;                 # Postgresql specific interface
#use DBD::Sybase;       # Sybase specific interface

$| = 1;

our($debug);
$debug = 0;

our($dbh, $dsn);

switch($dbtype) {
      case "DB2" {
            $dsn = "dbi:DB2:bench";
            $db_user="";
            $db_pass="";
      }
      case "Firebird" {
            $dsn =
"DBI:InterBase:db=/opt/firebird/bench.fdb;ib_dialect=3";
            $db_user="bench";
            $db_pass="bench";
      }
      case "MySQL" {
            $dsn = "DBI:mysql:database=bench;host=localhost;port=3306";
            $db_user="bench_user";
            $db_pass="bench1";
      }
      case "Oracle" {
            $dsn = "DBI:Oracle:";
            $db_user="bench";
            $db_pass="bench";
      }
      case "Postgresql" {
            $dsn = "DBI:Pg:";
            $db_user="bench";
            $db_pass="bench";
      }
      case "Sybase" {
            $dsn = "DBI:Sybase:server=VADER";
            $db_user="bench";
            $db_pass="benchpw";
      }
      default { die "DBD $dbtype not found."; }
}

our($n_children, $query_end_date, @query_start_date);
$n_children = 20;       # 20 processes, 1 per report
$query_end_date = fix_one_date("\"1/2/2008 0:00:00\"");
$query_start_date[1] = fix_one_date("\"1/1/2008 0:00:00\"");
$query_start_date[2] = fix_one_date("\"12/1/2007 0:00:00\"");
$query_start_date[3] = fix_one_date("\"1/1/2007 0:00:00\"");

# delete these
our(@male_names, @male_values, @female_names, @female_values,
@last_names,
      @last_values, @city_names, @city_values, @states);
our($INITIAL_CUSTOMERS, $INITIAL_TRANSACTIONS, $INITIAL_ITEMS,
      $INITIAL_STORES, $INITIAL_DEPARTMENTS);

$start_time = [gettimeofday];             # Start timer

# Run each child process in its own thread, for parallelism.
#
for($child_no=0; $child_no < $n_children; $child_no++) {
      $pid = fork();
      if($pid == 0) {   # in the child

            # start timer
            $report_start_time = [gettimeofday];

            # Login and create database handler
            #
            $dbh = DBI->connect($dsn, $db_user, $db_pass,
                  {PrintError => 1, RaiseError => 1, AutoCommit => 0})
                  or die "Database connection failed: $DBI::errstr";

            if($dbtype eq "Oracle") {
                  $dbh->do("alter session set " .
                        "nls_date_format='mm/dd/yyyy hh24:mi:ss'");
                  $dbh->commit();
            }

            switch($child_no) {
            case 0 {
                  ($report_file, $buffer) = store_profits(1);
                  }
```

```
case 1 {
      ($report_file, $buffer) = store_profits(2);
      }
case 2 {
      ($report_file, $buffer) = store_profits(3);
      }
case 3 {
      ($report_file, $buffer) = state_items(1);
      }
case 4 {
      ($report_file, $buffer) = state_items(2);
      }
case 5 {
      ($report_file, $buffer) = state_items(3);
      }
case 6 {
      ($report_file, $buffer) = department_revenues(1);
      }
case 7 {
      ($report_file, $buffer) = department_revenues(2);
      }
case 8 {
      ($report_file, $buffer) = department_revenues(3);
      }
case 9 {
      ($report_file, $buffer) = most_popular_items();
      }
case 10 {
      ($report_file, $buffer) = most_profitable_items();
      }
case 11 {
      ($report_file, $buffer) = state_customers(1);
      }
case 12 {
      ($report_file, $buffer) = state_customers(2);
      }
case 13 {
      ($report_file, $buffer) = state_customers(3);
      }
case 14 {
      ($report_file, $buffer) = city_customers(1);
      }
case 15 {
      ($report_file, $buffer) = city_customers(2);
      }
case 16 {
      ($report_file, $buffer) = city_customers(3);
      }
case 17 {
      ($report_file, $buffer) = top_customers(1);
      }
case 18 {
      ($report_file, $buffer) = top_customers(2);
      }
case 19 {
      ($report_file, $buffer) = top_customers(3);
      }
```

```
            default {
                  print "Shouldn't be in the default case.\n";
                  exit(-1);
                  }
            } # switch

            # end timer
            $report_elapsed_time = tv_interval($report_start_time);
            print "child $child_no elapsed time:
$report_elapsed_time\n";
            open FILE, ">reports/$report_file";
            print FILE $buffer;
            close FILE;

            # end the child
            exit(0);

      } # if in the child
} # for each report


$children_done = 0;
$child = 0;
while ($child != -1) {
      $child = wait();  # Returns -1 when no more children waiting.
      if($child != -1) {
            $children_done++;
      }
}

$elapsed_time = tv_interval($start_time); # End timer

print "$children_done finished $elapsed_time sec.\n";


# End of script.



# Convert date to DB2 format.
#
sub db2_convert_date {
      $old_date = shift;
      $old_date =~ m|"(\d+)/(\d+)/(\d+) (\S+)"|;
      return "'$3-$1-$2 $4'";
}


# Convert date to Firebird format.
#
sub firebird_convert_date {
      $old_date = shift;
      $old_date =~ m|"(.*)"|;
      return "'$1'";    # Just swap quotes for Firebird.
}
```

```
# Convert date to MySQL format.
#
sub mysql_convert_date {
      $old_date = shift;
      $old_date =~ m|"(\d+)/(\d+)/(\d+) (\S+)"|;
      return "\"$3-$1-$2 $4\"";
}


# Convert date to Oracle format.
#
sub oracle_convert_date {
      $old_date = shift;
      $old_date =~ m|"(.*)"|;
      return "'$1'";    # Just swap quotes for Oracle.
# We used the nls_date_format to tell Oracle the date format.
}


# Convert date to Postgresql format.
#
sub pg_convert_date {
      $old_date = shift;
      $old_date =~ m|"(.*)"|;
      return "'$1'";    # Just swap quotes for Postgresql.
}


# Convert date to Sybase format.
#
sub sybase_convert_date {
      $old_date = shift;
      return $old_date; # do nothing for Sybase.
}


# Identify which columns need date conversion and fix them.
#
sub fix_dates {
      my @cols = split /,/, shift;
      my @nums = split /,/, shift;
      switch($dbtype) {
            case "DB2" {
                  foreach $key (@nums) {
                        $cols[$key] = db2_convert_date($cols[$key]);
                  }
            }
            case "Firebird" {
                  foreach $key (@nums) {
                        $cols[$key] =
firebird_convert_date($cols[$key]);
                  }
            }
            case "MySQL" {
                  foreach $key (@nums) {
                        $cols[$key] = mysql_convert_date($cols[$key]);
                  }
            }
            case "Oracle" {
                  foreach $key (@nums) {
                        $cols[$key] = oracle_convert_date($cols[$key]);
                  }
            }
            case "Postgresql" {
                  foreach $key (@nums) {
                        $cols[$key] = pg_convert_date($cols[$key]);
                  }
            }
            case "Sybase" {
                  foreach $key (@nums) {
                        $cols[$key] = sybase_convert_date($cols[$key]);
                  }
            }
            default { die "DBD $dbtype"; }
      }
      return(join ",", @cols);
}


sub fix_one_date {
      $date = shift;
      switch($dbtype) {
            case "DB2" {
                  return db2_convert_date($date);
            }
            case "Firebird" {
                  return firebird_convert_date($date);
            }
            case "MySQL" {
                  return mysql_convert_date($date);
            }
            case "Oracle" {
                  return oracle_convert_date($date);
            }
            case "Postgresql" {
                  return pg_convert_date($date);
            }
            case "Sybase" {
                  return sybase_convert_date($date);
            }
            default { die "DBD $dbtype"; }
      }
} # fix_one-date


sub store_profits {
      $report_no = shift;

      $sum_club_member_qty = "sum(club_member_qty)";
      if($dbtype eq "Sybase" or $dbtype eq "Firebird" or $dbtype eq
"DB2") {
            $sum_club_member_qty = "(1.0 * sum(club_member_qty))";
      }
      if($dbtype eq "Postgresql") {
            $sum_club_member_qty = "float4(sum(club_member_qty))";
      }
      $sql = "
SELECT
      store_id as \"STORE_ID\", store_name as \"STORE_NAME\",
      sum(profit) as \"TOTAL_PROFIT\",
      sum(quantity) as \"N_ITEMS\", sum(qty_weight) as
\"TOTAL_WEIGHT\",
      $sum_club_member_qty / sum(quantity) as \"MEMBER_PCT\"
FROM (
      SELECT
            s.store_id, s.store_name,
            ti.discounted_price -
                  (i.wholesale_price * ti.quantity) as profit,
            ti.quantity, ti.quantity * i.weight as qty_weight,
            case when cm.customer_id is null then 0 else ti.quantity
                  end as club_member_qty
      FROM transactions t
      JOIN transaction_items ti ON (t.transaction_id =
ti.transaction_id)
      JOIN items i ON (ti.item_id = i.item_id)
      JOIN store_inventories si ON (t.store_id = si.store_id
            AND ti.item_id = si.item_id)
      JOIN stores s ON (si.store_id = s.store_id)
      JOIN customer_accounts ca ON (t.customer_id = ca.customer_id)
      LEFT JOIN club_members cm ON (ca.customer_id = cm.customer_id)
      WHERE
            t.tran_date < $query_end_date
            and t.tran_date >= $query_start_date[$report_no]
) sub1
GROUP BY
      store_id, store_name
ORDER BY
      sum(profit) desc
";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $filename = "store_profits_" . $report_no . ".out";
      $buffer = "";
      while($rowref = $sth->fetchrow_hashref) {
            %row = %$rowref;
            $buffer .= "$row{STORE_ID},$row{STORE_NAME},"
                  . "$row{TOTAL_PROFIT},$row{N_ITEMS},"
                  . "$row{TOTAL_WEIGHT},$row{MEMBER_PCT}\n";
      }
      print "Running store profits $report_no\n";
      return($filename, $buffer);

} # store_profits


sub state_items {
      $report_no = shift;

      $sql = "
SELECT
      cr.state as \"STATE\", sum(ti.quantity) as \"NO_ITEMS\",
      sum(t.total_weight) as \"TOTAL_WEIGHT\",
      sum(t.shipping_cost) as \"TOTAL_SHIPPING_COST\"
FROM transactions t
JOIN transaction_items ti ON (t.transaction_id = ti.transaction_id)
JOIN customer_accounts ca ON (t.customer_id = ca.customer_id)
JOIN customer_addresses cr ON (ca.customer_id = cr.customer_id AND
      ca.current_address = cr.sequence_number)
WHERE
      t.tran_date < $query_end_date
      and t.tran_date >= $query_start_date[$report_no]
GROUP BY
      state
";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $filename = "state_items_" . $report_no . ".out";
      $buffer = "";
      while($rowref = $sth->fetchrow_hashref) {
            %row = %$rowref;
            $buffer .= "$row{STATE},$row{NO_ITEMS},"
                  . "$row{TOTAL_WEIGHT},$row{TOTAL_SHIPPING_COST}\n";
      }
      print "Running state items $report_no\n";
      return($filename, $buffer);

} # state_items


sub department_revenues {
      $report_no = shift;

      $sql = "
SELECT
      department_id as \"DEPARTMENT_ID\", name as \"NAME\",
      sum(subtotal_revenue) as \"TOTAL_REVENUE\",
      sum(sub_no_items) as \"NO_ITEMS\", count(store_id) as
\"NO_STORES\"
FROM (
      SELECT
            d.department_id, d.name,
            sum(ti.discounted_price) as subtotal_revenue,
            sum(ti.quantity) as sub_no_items, t.store_id
      FROM transactions t
      JOIN transaction_items ti ON (t.transaction_id =
ti.transaction_id)
      JOIN items i ON (ti.item_id = i.item_id)
      JOIN departments d ON (i.department_id = d.department_id)
      WHERE
            t.tran_date < $query_end_date
            AND t.tran_date >= $query_start_date[$report_no]
      GROUP BY d.department_id, d.name, t.store_id
) subl
GROUP BY
      department_id, name
";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $filename = "department_revenues_" . $report_no . ".out";
      $buffer = "";
      while($rowref = $sth->fetchrow_hashref) {
            %row = %$rowref;
            $buffer .= "$row{DEPARTMENT_ID},$row{NAME},"
                  . "$row{TOTAL_REVENUE},$row{NO_ITEMS},"
                  . "$row{NO_STORES}\n";
      }
      print "Running department revenues $report_no\n";
      return($filename, $buffer);

} # department_revenues


sub most_popular_items {

      switch($dbtype) {
            case "DB2" {
                  $sql = "
SELECT
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.quantity) as \"TOTAL_QUANTITY\"
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.quantity) desc
FETCH FIRST 100 ROWS ONLY
";
            }
            case "Firebird" {
                  $sql = "
SELECT FIRST 100
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.quantity) as \"TOTAL_QUANTITY\"
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.quantity) desc
";
            }
            case "MySQL" {
                  $sql = "
SELECT
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.quantity) as \"TOTAL_QUANTITY\"
```

```
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.quantity) desc
LIMIT 100
";
            }
            case "Oracle" {
                  $sql = "
SELECT \"ITEM_ID\", \"NAME\", \"TOTAL_QUANTITY\" FROM (
SELECT
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.quantity) as \"TOTAL_QUANTITY\"
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.quantity) desc
) WHERE ROWNUM <= 100
";
            }
            case "Postgresql" {
                  $sql = "
SELECT
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.quantity) as \"TOTAL_QUANTITY\"
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.quantity) desc
LIMIT 100
";
            }
            case "Sybase" {
                  $sql = "
SET ROWCOUNT 100
SELECT
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.quantity) as \"TOTAL_QUANTITY\"
FROM transaction_items ti
```

```
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.quantity) desc
SET ROWCOUNT 0
";
            }
            default { die "unknown dbtype: $dbtype"; }
      } # end switch

      $sth = $dbh->prepare($sql);
      $sth->execute();
      $filename = "most_popular_items.out";
      $buffer = "";
      while($rowref = $sth->fetchrow_hashref) {
            %row = %$rowref;
            $buffer .= "$row{ITEM_ID},$row{NAME},"
                  . "$row{TOTAL_QUANTITY}\n";
      }
      print "Running most popular items\n";
      return($filename, $buffer);

} # most_popular_items


sub most_profitable_items {

      switch($dbtype) {
            case "DB2" {
                  $sql = "
SELECT
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) as \"TOTAL_PROFIT\"
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) desc
FETCH FIRST 100 ROWS ONLY
";
            }
            case "Firebird" {
                  $sql = "
SELECT FIRST 100
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) as \"TOTAL_PROFIT\"
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) desc
";
            }
            case "MySQL" {
                  $sql = "
SELECT
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) as \"TOTAL_PROFIT\"
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) desc
LIMIT 100
";
            }
            case "Oracle" {
                  $sql = "
SELECT \"ITEM_ID\", \"NAME\", \"TOTAL_PROFIT\" FROM (
SELECT
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) as \"TOTAL_PROFIT\"
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) desc
) WHERE ROWNUM <= 100
";
            }
            case "Postgresql" {
                  $sql = "
SELECT
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) as \"TOTAL_PROFIT\"
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) desc
LIMIT 100
";
            }
            case "Sybase" {
                  $sql = "
SET ROWCOUNT 100
SELECT
      i.item_id as \"ITEM_ID\", i.name as \"NAME\",
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) as \"TOTAL_PROFIT\"
FROM transaction_items ti
JOIN items i ON (ti.item_id = i.item_id)
JOIN transactions t ON (ti.transaction_id = t.transaction_id)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[3]
GROUP BY
      i.item_id, i.name
ORDER BY
      sum(ti.discounted_price -
            (i.wholesale_price * ti.quantity)) desc
SET ROWCOUNT 0
";
            }
            default { die "unknown dbtype: $dbtype"; }
      } # end switch

      $sth = $dbh->prepare($sql);
      $sth->execute();
      $filename = "most_profitable_items.out";
      $buffer = "";
      while($rowref = $sth->fetchrow_hashref) {
            %row = %$rowref;
            $buffer .= "$row{ITEM_ID},$row{NAME},"
                  . "$row{TOTAL_PROFIT}\n";
      }
      print "Running most profitable items\n";
      return($filename, $buffer);

} # most_profitable_items
```

```
sub state_customers {
      $report_no = shift;

      $sql = "
SELECT
      state as \"STATE\", count(*) as \"NO_CUSTOMERS\"
FROM transactions t
JOIN customer_accounts ca ON (t.customer_id = ca.customer_id)
JOIN customer_addresses cr ON (ca.customer_id = cr.customer_id AND
      ca.current_address = cr.sequence_number)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[$report_no]
GROUP BY
      state
ORDER BY
      count(*) desc
";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $filename = "state_customers_" . $report_no . ".out";
      $buffer = "";
      while($rowref = $sth->fetchrow_hashref) {
            %row = %$rowref;
            $buffer .= "$row{STATE},$row{NO_CUSTOMERS}\n";
      }
      print "Running state customers $report_no\n";
      return($filename, $buffer);

} # state_customers


sub city_customers {
      $report_no = shift;

      $sql = "
SELECT
      city as \"CITY\", state as \"STATE\", count(*) as
\"NO_CUSTOMERS\"
FROM transactions t
JOIN customer_accounts ca ON (t.customer_id = ca.customer_id)
JOIN customer_addresses cr ON (ca.customer_id = cr.customer_id AND
      ca.current_address = cr.sequence_number)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[$report_no]
GROUP BY
      state, city
ORDER BY
      count(*) desc
";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $filename = "city_customers_" . $report_no . ".out";
      $buffer = "";
      $count = 0;
      while($rowref = $sth->fetchrow_hashref) {
            %row = %$rowref;
            $buffer .= "$row{CITY},$row{STATE},$row{NO_CUSTOMERS}\n";
            $count++;
            last if($count == 100);
      }
      print "Running city customers $report_no\n";
      return($filename, $buffer);

} # city_customers


sub top_customers {
      $report_no = shift;

$sql = "
SELECT
      ca.customer_id as \"CUSTOMER_ID\", first_name as \"FIRST_NAME\",
      last_name as \"LAST_NAME\", city as \"CITY\", state as \"STATE\",
      sum(total) as \"TOTAL_SPENT\"
FROM transactions t
JOIN customer_accounts ca ON (t.customer_id = ca.customer_id)
JOIN customer_addresses cr ON (ca.customer_id = cr.customer_id AND
      ca.current_address = cr.sequence_number)
WHERE
      t.tran_date < $query_end_date
      AND t.tran_date >= $query_start_date[$report_no]
GROUP BY
      ca.customer_id, first_name, last_name, city, state
ORDER BY
      sum(total) desc
";
      $sth = $dbh->prepare($sql);
      $sth->execute();
      $filename = "top_customers_" . $report_no . ".out";
      $buffer = "";
      while($rowref = $sth->fetchrow_hashref) {
            %row = %$rowref;
            $buffer .= "$row{CUSTOMER_ID},$row{FIRST_NAME},"
                  . "$row{LAST_NAME},$row{CITY},"
                  . "$row{STATE},$row{TOTAL_SPENT}\n";
            $count++;
            last if($count == 100);
      }
      print "Running top customers $report_no\n";
      return($filename, $buffer);

} # top_customers
```