Summer 2013

A Study of Mongodb and Oracle in an E-Commerce Environment

Aaron Ploetz
Regis University

Regis University
College for Professional Studies Graduate Programs
Final Project/Thesis

Disclaimer

A STUDY OF MONGODB AND ORACLE IN AN E-COMMERCE ENVIRONMENT

A THESIS

SUBMITTED ON 03 JULY 2013

TO THE DEPARTMENT OF INFORMATION TECHNOLOGY

OF THE SCHOOL OF COMPUTER & INFORMATION SCIENCES

OF REGIS UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF MASTER OF SCIENCE IN
SOFTWARE ENGINEERING AND DATABASE TECHNOLOGIES

BY

_____
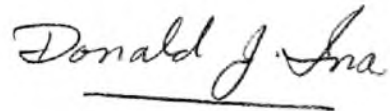
Aaron Ploetz
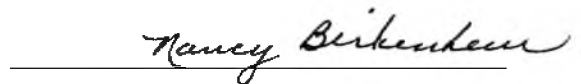
APPROVALS

_____

Darl Kuhn, Thesis Advisor

_____

Don Ina, Faculty of Record

_____

Nancy Birkenheuer, Program Coordinator

Abstract

As worldwide e-commerce expands, businesses continue to look for better ways to meet their evolving

needs with web solutions that scale and perform adequately. Several online retailers have been able to

address scaling challenges through the implementation of NoSQL databases. While architecturally

different from their relational database counterparts, NoSQL databases typically achieve performance

gains by relaxing one or more of the essential transaction processing attributes of atomicity,

consistency, isolation, and durability (ACID). As with any emerging technology, there are both critics

and supporters of NoSQL databases. The detractors claim that NoSQL is not safe and is at a greater

risk for data loss. On the other hand, its ardent defenders boast the performance gains achieved over

their relational counterparts. This thesis studies the NoSQL database known as "MongoDB," and

discusses its ability to support the growing needs of e-commerce data processing. It then examines

MongoDB's raw performance (compared to Oracle 11g R2, a relational database) and discusses its

adherence to ACID.

Table of Contents

List of Figures

List of Tables

List of Abbreviations Used

ACID – Transaction which support Atomicity, Consistency, Isolation, and Durability (ACID).

Relational database management systems are known for their support of ACID transactions.

ACIDity – A somewhat arbitrary method of describing the level that database transactions adhere to

ACID properties.

AMD – Advanced Micro Devices, Inc. A U.S.-based company which is chiefly known for

manufacturing microprocessors, motherboard chipsets, and semiconductors.

API – Application Programmer Interface. A library written to abstract low-level data access tasks with

the goal of increasing developer productivity.

BASE – Basically Available Soft-state Eventual consistency. BASE is considered to be the antithesis

of ACID.

BSON – Binary JSON. Data serialization through the binary encoding of JSON documents, used by

MongoDB to store data.

CAP – Consistency, Availability, and Partition tolerance make up the three attributes of Brewer's CAP

Theorem.

CRUD – Create Read Update Delete.

CSV – Comma Separated Value. Comma Separated Value files are a common way of organizing data

in export files.

DBA – Database Administrator. An individual who is responsible for the configuration and

operational performance of a database.

DDL – Data Definition Language. A subset of SQL used to define database tables, indexes, and other

objects.

ER – Entity Relationship. Typically used to describe an entity relationship diagram, which maps out the relationships between the database entities.

ERP – Enterprise Resource Planning. An ERP system is an application which handles a business' customer relationships, supply chain and/or employee data. Major ERP vendors include Oracle and SAP.

FOSS – Free and Open Source Software. Software that is free of charge and provides the source code to allow it to be changed by its user.

IEEE – Institute of Electrical and Electronics Engineers. A professional society promoting standards and technological advancement.

JSON – JavaScript Object Notation. A notation commonly used for serializing object data and/or passing data between distributed services.

LAN – Local Area Network.

MIT – Massachusetts Institute of Technology.

ms – Milliseconds.

NoSQL – A label for non-RDBMS, big-data databases that literally means "Not only SQL," in reference to the Standard Query Language used by RDBMSs.

ns – Nanoseconds.

PCRE – Perl-Compatible Regular Expressions. A regular expression API implemented in both PHP and Oracle.

PHP – Personal Home Page tools. PHP is a popular language (Lerdof, Tatrow, 2002) developed by Rasmus Lerdof in 1995, which allows developers to easily add logic to HTML-based web pages.

POSIX – Portable Operating System Interface for UNIX (Casteel, 2007). Essentially, it is a set of IEEE standards which allows for the portability and compatibility of software between different platforms.

IT – Information Technology.

RAM – Random Access Memory.

RDBMS – Relational Database Management System.

RMAN – Oracle Recovery Manager. It is (Kuhn, 2010, p. 457) "Oracle's flagship backup and recovery (B&R) tool."

SKU – Stock Keeping Unit. An identifier typically used by retailers to refer to a specific product number.

SQL – Structured Query Language. A scripting language used by relational database management systems.

SSL – Secure Socket Layer. A widely-used authentication protocol for secure network communications.

Chapter 1 - Introduction

NoSQL databases have proven to be a viable solution for some unique scaling scenarios. However, the idea of implementing a NoSQL solution (instead of a relational database) seems premature to many information technology organizations. Often, experienced professionals will decide to "live with" a process that is lengthy or slow due to an inability to scale appropriately.

Adding more confusion is NoSQL's status as a known "buzzword." This has the effect of people seeking it out when it may not be the best solution. There have been many instances of early adopters who do not really understand NoSQL technology (Banker, 2010), who have run into issues trying to implement a relational model with it.

InfoWorld columnist Andrew Oliver illustrates this confusion as he describes one individual who attempted (unsuccessfully) to implement MongoDB. He writes that (Oliver, 2012, p. 1) "if you're working on a tiny application that doesn't have high scalability requirements and you're familiar with PostgreSQL, why not use PostgreSQL?" Oliver's point is that if a developer or organization has in-house knowledge of a particular tool, and that tool will accomplish the task at-hand, then they should use it. Later on, he adds (Oliver, 2012, p.1) "The lesson to be learned here is: Don't solve a simple problem with a completely unfamiliar technology and apply it to use cases it isn't especially appropriate for."

Are there problems that can be more appropriately solved with a NoSQL database instead of a relational database management system (RDBMS)? Can NoSQL databases out-perform relational databases in some scenarios? Can NoSQL provide some semblance of ACID transactions? These are questions that I will attempt to answer in this thesis.

What Makes NoSQL Different?

NoSQL database technology is premised on the work done by Dr. Eric Brewer on distributed systems. In his 1999 paper (co-authored with Armando Fox) "Harvest, Yield, and Scalable Tolerant

Systems", Brewer (Brewer, Fox, 1999) introduced what he called the "CAP Principle." Essentially, every distributed system strives to address three primary concerns:

1. Consistency – A system that has (Brewer, Fox 1999, p. 2) strong consistency" or "single-copy ACID consistency; by assumption a strongly-consistent system provides the ability to perform updates."

2. Availability – (Brewer, Fox 1999) High availability is achieved if a read request can always reach and return data from a replica.

3. Partition Tolerance – The ability to easily add replicas of the data to allow for horizontal scaling.

The key point of Brewer's CAP Principle (later known as both "Brewer's CAP Conjecture" and "Brewer's CAP Theorem") indicated that it was impossible for a distributed system to fully satisfy all of these conditions. MIT researchers Nancy Lynch and Seth Gilbert verified this in their 2002 paper, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." They found that (Gilbert, Lynch, 2002, p. 11) "it is impossible to reliably provide atomic, consistent data when there are partitions in the network." They agreed with Brewer's conclusion, that it was possible to (Gilbert, Lynch, 2002, p.11) "achieve any two of the three properties: consistency, availability, and partition tolerance."

Given that a distributed system (e.g.: database) could be expected to satisfy any two of points of the CAP Principle, Brewer devised a list of characteristics hereafter referred to as "CAP Configurations." These can be used to identify the architectural strengths of various database systems:

1. CA – (Brewer, Fox, 1999) "Databases that provide distributed transactional semantics can only do so in the absence of a network partition separating server peers." Relational database

systems are considered to be "CA" databases, as they have the ability to provide strong ACID

transactions. Partition tolerance is a secondary consideration, and is typically achieved through

replication, clustering or fail-over strategies.

2. AP – This designation describes databases that are both highly available and partition tolerant.

    With consistency as a secondary consideration, most of these NoSQL databases (such as

    Cassandra, Riak, and CouchDB) operate under the assumption of optimistic or (Browne, 2009)

    eventual consistency. However, the performance benefits achieved by sacrificing consistency

    (Yu, Vahdat, 2000, pp. 1-2) "comes at the expense of an increasing probability that individual

    accesses will return inconsistent results, e.g., stale/dirty reads, or conflicting writes."

3. CP – NoSQL databases achieve ACID-like transactions to their partitions through locking

    strategies. These types of systems (such as MongoDB) sacrifice high-availability to ensure

    "strong" data consistency across all replicas/nodes, and (Brewer, Fox, 1999, p. 2) "avoid the

    risk of introducing merge conflicts (and thus inconsistency)."

Essentially, NoSQL databases are architecturally different from relational databases because

they are designed to reap the read and write performance benefits of partition tolerance (horizontal

scaling), while leaving either consistency or availability up for negotiation. Knowing this difference is

paramount to understanding the situations where a NoSQL database may be a better fit than a RDBMS.

Why is ACID Important?

One common way to describe relational database transactions is to say that they contain

properties of atomicity, consistency, isolation, and durability. These transaction properties are

collectively known as ACID. The ability to provide ACID transactions is one of the main reasons

many businesses utilize RDBMSs. As author Joan Casteel illustrates in her 2007 book "Oracle 10g

SQL," an example that makes a good case for ACID transactions (Casteel, 2007) is a banking

transaction.

Assume a bank transaction in which $500 is withdrawn from one account, split in half and $250 is deposited into two other accounts. If the bank's data center should suffer a power failure after the money is withdrawn, but before the other two transactions reach completion, is that $500 lost? If the transactions were completed on a database that supports ACID transactions, then (Casteel, 2007, p. 142) "the save will not occur until all three actions are entered."

It should be noted that (Banker, 2012, p. 256) "MongoDB doesn't provide ACID guarantees over a series of operations, and no equivalent of RDBMSs' BEGIN, COMMIT, and ROLLBACK semantics exists." However, MongoDB does support (Banker, 2012, p. 256) "atomic, durable updates on individual documents and consistent reads." The take-away from this: if you have a multi-operation transaction that decrements from one property and increments another (on a single document), then MongoDB can ensure atomicity and durability in this case. On the other hand, MongoDB cannot ensure atomicity (or durability) on a multi-operation transaction that updates properties on separate or multiple documents.

With ACID transaction support defined, it is time to ask next logical question. "Why would anyone choose to not implement ACID transactions?" After all, the reasons to use ACID transactions seem to be grounded in data safety and common sense.

In some scenarios, e-commerce retailers may allow certain non-critical data to go inconsistent or "stale" with the majority of nodes, for short periods of time. Assume a scenario where a customer places an order on a website, and it makes its way through their ERP system. After a while, it may be sent back to the order history database (in batch), and made available for query by a web user. If the order history database nodes are updated every hour, then a partial or failed update is not critical to a business transaction. In this case, an ACID transaction is not required as there is a good chance

(assuming that the issue which caused the original failure has been rectified) that the order history data will re-update successfully within the next hour.

The benefit to be gained (in the previous scenario) is that of raw performance. Large internet retailers may have hundreds of thousands (or even millions) of customers. A database containing order history records for prior years may grow big rather quickly. The absence of ACID transactions allows NoSQL databases to quickly serve this type of data across multiple nodes.

Other times, going without ACID guarantees is not so much a choice as it is a way to deal with system failure. In the case of Amazon, which has (Decandia et al, 2007, p. 205) "tens of thousands of servers" spread across the world; there is almost always some component in a state of failure. To counter that, Amazon's Dynamo NoSQL database (Decandia et al, 2007, p. 205) needed "to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance." In this case, as long as the data is consistent across a majority (or preconfigured quorum) of nodes, the transaction is considered to be successful.

Purpose of this Study

This study will generate data based on e-commerce use cases to ascertain levels of performance and data integrity on both MongoDB and Oracle database systems. A testing framework will simulate an e-commerce website. As a part of the testing framework, a series of common e-commerce functions will be written to perform operations against each database.

Test data will be generated to simulate customers, addresses (of customers) and products. The data will then be loaded into each database (MongoDB and Oracle 11g), so that each will have the same product, customer, and address data. The database instances will be running on Linux machines with similar hardware configurations. Next, a series of experiments will be run using the aforementioned e-commerce functions of the testing framework. Statistics will be tracked for

performance of create, read, update, and delete (also known as CRUD) transactions typical for an e-commerce website.

Some of the experiments will be focused on testing the ACID transaction properties. The Oracle 11g R2 database is assumed to be fully ACID compliant, so it will not be subject to ACID tests. However, tests against MongoDB will be scrutinized for its adherence to ACID properties.

One goal of this study is to determine how well MongoDB and Oracle 11g R2 match-up to each other in the context of e-commerce usage. Many e-commerce database back-ends are built on relational databases and still manage to provide adequate performance. They do this while providing ACID transaction guarantees, and a feature-rich toolset for their developers and users. While MongoDB may not have this same level of features, discovering if it can support a favorable trade-off of features for raw performance is an aim of this study.

Please note that this paper assumes that the reader has an intermediate to advanced understanding of database and software development terminology. All significant terms used in this paper have been defined in Appendix A "Definition of Terms." A list (and definition) of all significant acronyms used can also be found in the "List of Abbreviations" chapter.

Research Questions and Hypotheses

There are three fundamental questions which have driven the direction of this study. These questions help to underscore the differences between Oracle and MongoDB, and (by proxy) RDBMS and NoSQL technologies. These questions are:

1. Between MongoDB and Oracle, which will perform faster e-commerce-based transactions?

2. Between MongoDB and Oracle, which will provide greater data integrity (based on ACID standards)?

3. Is there an acceptable level of compromise between performance and data integrity to allow for an optimal e-commerce solution on MongoDB?

I hypothesize the following answers, respectively:

1. MongoDB will out-perform Oracle in terms of raw read and write speeds.

2. MongoDB will fail to adequately negotiate multiple ACID tests.

3. An appropriate trade-off of ACID for performance will be able to be made, to justify the use MongoDB in an e-commerce environment.

Limitations of this Study

My original thought was to compare NoSQL databases to RDBMSs. However, that topic was quickly determined to be too broad. NoSQL databases vary dramatically (even when compared to each other) from an architectural perspective. To assume that experiments done with MongoDB would reflect those done with Cassandra or CouchDB would be erroneous. Testing multiple NoSQL databases would be challenging and time-consuming, therefore, a single NoSQL database had to be chosen.

Also the original implementation of this study was to be on "traditional data processing environments." That statement is too vague to do a valuable study on, and had to be refined. Due to my experience and qualifications, the use cases for this study were chosen to relate to the e-commerce industry.

MongoDB (developed by 10gen, Inc.) has been chosen as the NoSQL subject for this study. It has been advocated for use in e-commerce by several authors, including "MongoDB in Action" author Kyle Banker and Forward Systems founder Eric Ingram. In their book "The Definitive Guide to MongoDB," authors Eelco Plugge, Peter Membrey, and Tim Hawkins mention that (Membrey, et-al,

2010, ch. 1) "for many things (such as building a web application), MongoDB can be an awesome tool for implementing your solution."

While I do have experience in developing for and maintaining Cassandra NoSQL databases, Cassandra was determined to not be the best choice for this study. As author Kristov Kovacs noted, the best scenario for using Cassandra is if (Kovacs, 2012, para. 20) "you write more than you read." While a typical e-commerce back-end database does plenty of writing (customer orders, product updates, etc...), most of the queries handled will be reads to support guided navigation, product and customer lookups. Also, Cassandra is known to have an AP CAP Configuration, which means that it sacrifices consistency. This was determined not to be ideal, as there are certain parts of e-commerce systems (like orders and payments) that will require consistency across partitions.

Oracle 11g was chosen to represent the RDBMS side of this study. Oracle is widely considered to be the front-runner in the current RDBMS market. ServerWatch's Kenneth Hess (Hess, 2010, p. 3) puts Oracle at the top of the list for enterprise databases, adding that "Oracle's name is synonymous with enterprise database systems." Craig S. Mullins of The Data Administration Newsletter (TDAN) indicates that (Mullins, 2011, para. 3) Oracle is the "clear leader" in the RDBMS realm with a 48.1% market share.

It should also be noted that Oracle provides a solution for horizontal scaling with Oracle Real Application Clusters (RAC). The Oracle model for this study will not be subjected to the ACID tests, and will not be required to persist data on distributed nodes. Therefore, the implementation of Oracle in a RAC was determined to be out of scope for this project.

Chapter 2 - Review of Existing Literature

Introduction

Oracle's RDBMS has been Oracle's flagship product for more than 30 years. By contrast, production-ready versions of MongoDB were not released until 2010; it is still considered to be a new product. This distinction also makes it a difficult subject upon which to find good resources. Additionally peer-reviewed, scientific papers about MongoDB are even rarer. However, papers and journals covering basic NoSQL concepts do exist.

This review will cover literature addressing major aspects of NoSQL architecture, and especially those which are directly related to MongoDB and this project. This includes literature which helped to address the three hypothesized research questions, as well as works that assisted in the design of the lab environment required for this study. It should be noted that the majority of existing works examined will be focused on MongoDB and other NoSQL architectures. While a functioning Oracle 11g R2 database is certainly an integral part of this project, Oracle architecture is well-known and also not the central focus of examination.

Brewer's CAP Theorem

As mentioned in the introduction chapter, the concepts behind NoSQL database architecture are based on Dr. Eric Brewer's work. Early research quickly introduced me to the concept of "Brewer's CAP Theorem" and eventually, Brewer and Dr. Armando Fox's 1999 paper "Harvest, Yield, and Scalable Tolerant Systems." In this paper, Brewer discussed the challenges of providing strong consistency and high availability in a distributed system. This paper signified the introduction of the CAP Theorem, which Brewer described as his "CAP Principle." The CAP Principle (Brewer, Fox, 1999, p. 2) "makes explicit trade-offs in designing distributed infrastructure applications." Essentially it defined Consistency, Availability, and Partition Tolerance as desirable attributes of distributed

systems, but stated that it is only realistic to fulfill two of those at any given time.

Following Brewer's paper, I discovered Dr. Seth Gilbert and Dr. Nancy Lynch's 2002 paper entitled "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." Gilbert and Lynch offered a series of proofs in an asynchronous network model to demonstrate the tradeoffs between availability and consistency. Using this model, they concluded that Brewer was correct in that it is (Gilbert, Lynch 2002, p.11) "impossible to reliably provide atomic, consistent data when there are partitions in the network." These papers are significant in that they laid the foundation of the concepts by which all NoSQL databases are designed.

NoSQL Tradeoffs for Consistency, Performance, and Availability

Another significant work in the NoSQL world is "Design and Evaluation of a Continuous Consistency Model for Replicated Services" by Dr. Haifeng Yu and Dr. Amin Vahdat. This paper further illustrated the trade-offs between consistency, availability, and performance. Additionally Yu and Vahdat introduced a continuous consistency model, in which they demonstrated the ability to (Yu, Vahdat, 2000) provide a configurable level of consistency in a data store with remote replicas.

One of the more-recent works to break new ground in the NoSQL world is Amazon's 2007 "Dynamo: Amazon's Highly Available Key-value Store." This paper was written by several Amazon researchers including (but not limited to) Giuseppe DeCandia, Peter Vosshall, and Werner Vogels. In addition to providing insight into the architecture behind one of the largest e-commerce operations in the world, this paper described the evolution of Amazon's Dynamo NoSQL database, and the types of availability issues that it was architected to solve.

While also designed to scale to serve (Decandia, et-al, 2007, p.205) "tens of millions of customers at peak times using tens of thousands of servers," Amazon Dynamo delivers high-availability by treating failure as a normal state. This paper also discussed the decision points that led

to the designing of Dynamo to operate with "weaker" replica consistency. These concepts (among others) led to the creation of other Dynamo-based NoSQL products such as Cassandra and Riak. While not directly pertaining to the architecture behind MongoDB, this paper is significant to this study in that Dwight Merriman and the original developers of MongoDB had to answer these same types of questions in their design.

MongoDB Technical Guidance

This section will cover some of the more technical MongoDB materials available. One of the central aspects of this project involved the planning, installation, and operation of MongoDB databases. In addition, these sources were also referenced in structuring the application calls to the MongoDB Java API.

Not surprisingly, much of the existing materials are written by current and former 10gen employees. In terms of working with MongoDB specifically, former 10gen engineer Kristina Chodorow published three works which address the perspectives of application development for and database administration of MongoDB. This study utilized the following works by Ms. Chodorow: "MongoDB: The Definitive Guide," "50 Tips and Tricks for MongoDB Developers," and "Scaling MongoDB."

Kristina Chodorow and (former 10gen engineer) Michael Dirolf's "MongoDB: The Definitive Guide" provided an introduction to the core concepts of MongoDB. It described topics such as quick installation, document structure and behavior, querying, indexing, and aggregation. Later chapters covered subjects such as sharding, replication, and administration. This work proved to be instrumental to writing the early prototypes of the testing framework and data loaders.

Chodorow's "50 Tips and Tricks for MongoDB Developers" serves as a quick reference for application developers working with MongoDB as a data store. The book is broken up into sections

pertaining to application design, implementation, organization, data safety and consistency, and administration. The examples provided in the application design and implementation sections proved useful to the design of the MongoDB data store and code for this project.

"Scaling MongoDB" (also by Kristina Chodorow) is a reference for MongoDB DBAs working with a sharded cluster. This book offered details on the setup, administration, and behavior of MongoDB shards. Chodorow also discussed how to distribute and balance your data, emphasizing the importance of choosing a good (Chodorow, 2011b) shard key. Examination of this source also provided insight into how to test MongoDB for specific ACID properties.

During my research on MongoDB use cases and schema designs, I happened to find (former 10gen engineer) Kyle Banker's article titled "MongoDB and E-commerce" (Banker, 2010). This article described modeling an entire e-commerce system using MongoDB. While Banker would later publicly retract the assertions made about the feasibility of running the order/payment operations in MongoDB (due to lack of atomic transactions), his design strategies proved to be helpful for this project.

Reading through Banker's "MongoDB and E-commerce" also led me to his 2012 book titled "MongoDB In Action." While discussing the architecture and administration of MongoDB, Banker specifically described e-commerce schemas, including a section called "Designing an e-commerce data model." In this section, he presented a detailed product document model, as well as a separate model for category documents. These models directly influenced the e-commerce data models used in this thesis.

Summary

These works demonstrated their relevance to this subject by illustrating some of the problems introduced by databases with network partitions. They also bring to light some of the methods used to

solve those problems, essentially providing an avenue to understand why NoSQL databases are

designed the way that they are. Many of the MongoDB-specific sources were relied upon heavily

during the construction of the testing framework. The concepts introduced in this chapter will be

referred to hereafter, as we explore some of the advantages and challenges of using MongoDB in an e-

commerce environment. Please note that additional information on this and other works which have

influenced this thesis can be found in Appendix B: Annotated Bibliography.

Chapter 3 - Research Methodology

This chapter describes the research methodologies used for this study. It defines the research methods and how they will help to answer the posed research questions. Also included are sections detailing the lab hardware and software, the database configuration and schema design, as well as strategies used for benchmarking database performance.

Research Approach

This study will use a quantitative approach to answer the three research questions posed in the introduction:

1. Between MongoDB and Oracle, which will perform faster e-commerce based transactions?

2. Between MongoDB and Oracle, which will provide greater data integrity (based on ACID standards)?

3. Is there an acceptable level of compromise between performance and data integrity to allow for an optimal e-commerce solution on MongoDB?

Answers to the first question will be the result of performance statistics obtained by measuring the application-level performance of calls to both MongoDB and Oracle 11g R2. The answers to the second question will be determined by the pass/fail status of ACID tests on MongoDB. The third question will be answered by analyzing data from the first and second questions.

Description of Lab Hardware

This section details the hardware used to perform the performance experiments. It should be noted that I performed this study with my own resources, and without access to enterprise-grade hardware. Therefore this experiment utilized two AMD-based machines, detailed below:

System #1: AMD Athlon 64 X2 6400 3.2GHz Dual-Core Processor, 4GB DDR2 SDRAM,

7200rpm 260GB hard disk, Ubuntu (64-bit server edition) 10.04.

System #2: AMD Athlon 64 X2 4600 2.4GHz Dual-Core Processor, 4GB DDR2 SDRAM,

7200rpm 500GB hard disk, Ubuntu (64-bit server edition) 10.04.

These systems supported two different configurations (as shown in Figure 3-1). Only one

database was running at any given time. This prevented the systems from having to divide resources

between two concurrently-running database instances.

Hardware Configuration #1.

The first configuration (Figure 3-1) was designed for the Oracle 11g R2 experiments. System

#1 functioned as the application server, and sent requests to the Oracle database on System #2 via a

LAN connection. All tests for Oracle 11g R2 were run against one standalone-database server.

Configuring Oracle for replication across multiple servers (RAC cluster) was determined to be beyond

the scope of this project (partition tolerance is not considered to be a strength of relational databases, so

there is no need to test it).
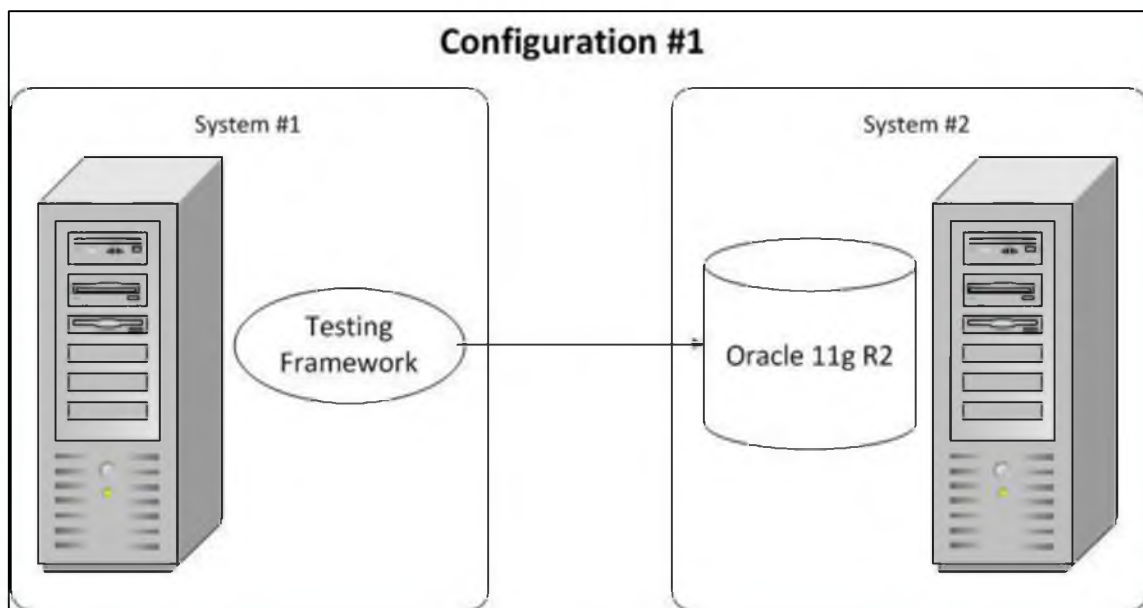
Figure 3-1. Diagram of the Oracle 11g R2 hardware configuration.

Hardware Configuration #2.

While similar, Configuration #2 differed in that a MongoDB slave or "SECONDARY"

replication server also ran on System #1 (as shown in Figure 3-2), alongside the testing framework. It

executed transactions against the MongoDB database on System #2. Data from the MongoDB

"PRIMARY" was then replicated back to the MongoDB "SECONDARY" on System #1.
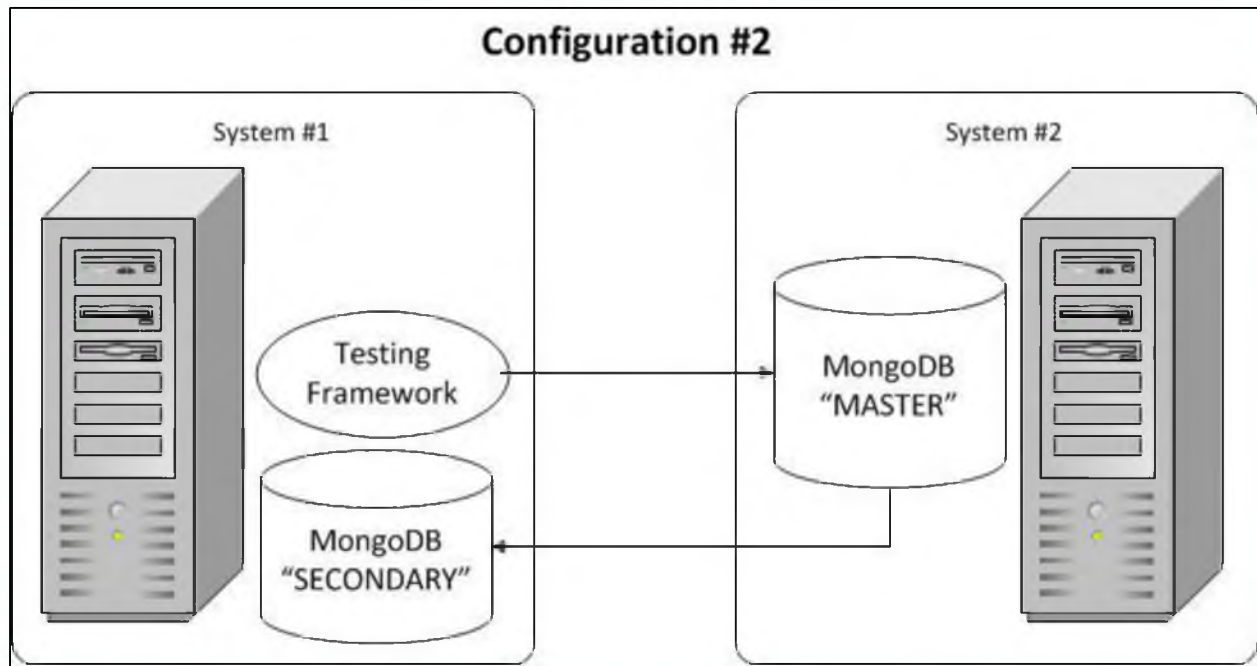
Figure 3-2. Diagram of the MongoDB hardware configuration.

Operating System Environment.

A decision was made early on in this study to have all systems running the Ubuntu version of

the Linux operating system. While I do have extensive experience with both commercial Linux and

Windows operating systems, it was more cost-effective to build the systems running a FOSS operating

system. For this environment I have selected Ubuntu 10.04, a Debian-based Linux operating system.

I chose Ubuntu 10.04 due to my experience with that version, the maturity of the release, and its

cost (free). It should be noted that Oracle does not officially support enterprise database installations

on Ubuntu. However I was unable to discover any technical reasons indicating that Oracle would not

install or that it would perform poorly on a system running Ubuntu. And despite the level of official

documentation on installing Oracle 11g R2 on the Ubuntu operating system, I was able to get it to

install and run with minimal issues.

Description of Lab Software

A testing framework was written in Java which ran on System #1 and interfaced with the

database on System #2. Java was chosen as the language for the testing framework due to my

experience in that language, as well as the popularity and maturity of the database software APIs. The

specific API Java Archive (JAR) files and versions used for each database are listed here:

- MongoDB: mongo-2.7.3.jar

- Oracle 11g R2: ojdbc14.jar

To begin, the database software was installed on System #2. In the case of MongoDB, version

2.0 was downloaded, unzipped and initialized in about a half an hour. As an additional challenge,

MongoDB 2.2 was released on August 28th of 2012, requiring an upgrade of the database software

prior to the "production" run of the experiments. In the case of Oracle, version 11g R2 Enterprise

Edition was downloaded and installed.

Use of the Term "Schema"

It should be noted that there is more than one common definition for "schema." Typically,

schema is used as to describe the structure of tables and how they relate to one another. There will also

be references in this paper as to how MongoDB is "schema-less" or how it has a "flexible" schema.

This is simply meant to imply that MongoDB does not enforce any kind of document (row) structure

within its collections (tables).

Within the context of an Oracle database however, the use of "schema" indicates a (Casteel,

2007) collection of database objects which belong to a single user. In this way, "schema" means more

than just the design and interrelationships of the tables. It encompasses tables, indexes, triggers and

many other types of database objects.

Given the subject matter of this paper, there will be several uses of the word "schema." For purposes of clarification, it should be noted that this paper will use "schema" simply to indicate the design of the tables as well as the relationships between them. If "schema" is used hereafter to refer to an "Oracle schema," this distinction will be clearly made. Please note that this and other terminology differences between Oracle and MongoDB have been outlined in Appendix C, "Oracle and MongoDB Comparison of Terms."

Database Schema Design

A preliminary (normalized) e-commerce database schema was then developed for the Oracle 11g R2 instance (Figure 3-3 is an entity relationship diagram which details the schema used in this configuration). A "products" table was required, as well as a related table for multiple price points. Additional design characteristics for the MongoDB schema were taken from Kyle Banker's "MongoDB and E-commerce" (Banker, 2010) which demonstrated embedded documents in lieu of complex JOIN operations. Additionally, a table was created to handle the product hierarchy, where each product belongs to a hierarchy consisting of a top-level category and a product group.

The database also needs to keep information on customers, and the "customers" table was modeled (Grobler, 2010) to contain data pertaining to the customer's name, email, and password. This project's customer data model also had to account for the likelihood that customers will have multiple addresses (ship-to, bill-to, etc…). This led to the creation of the addresses table with a one-to-many relationship from customers to addresses. The customers entity in the MongoDB schema contained the addresses as documents embedded inside the customer documents.

Finally, tables were created to keep track of data relating to e-commerce orders (customers, products ordered, prices paid, etc…). The model used contained entities for both Orders and OrderItems because (Song, Whang, 2000) each order can contain several items, demonstrating a one-to-many relationship. For the MongoDB schema, the items purchased on each order were added

(Banker, 2010) as embedded documents. Additional technical details on the database schema used for

the Oracle 11g R2 tests can be found in Appendix D "E-commerceDB Schema (Oracle)."
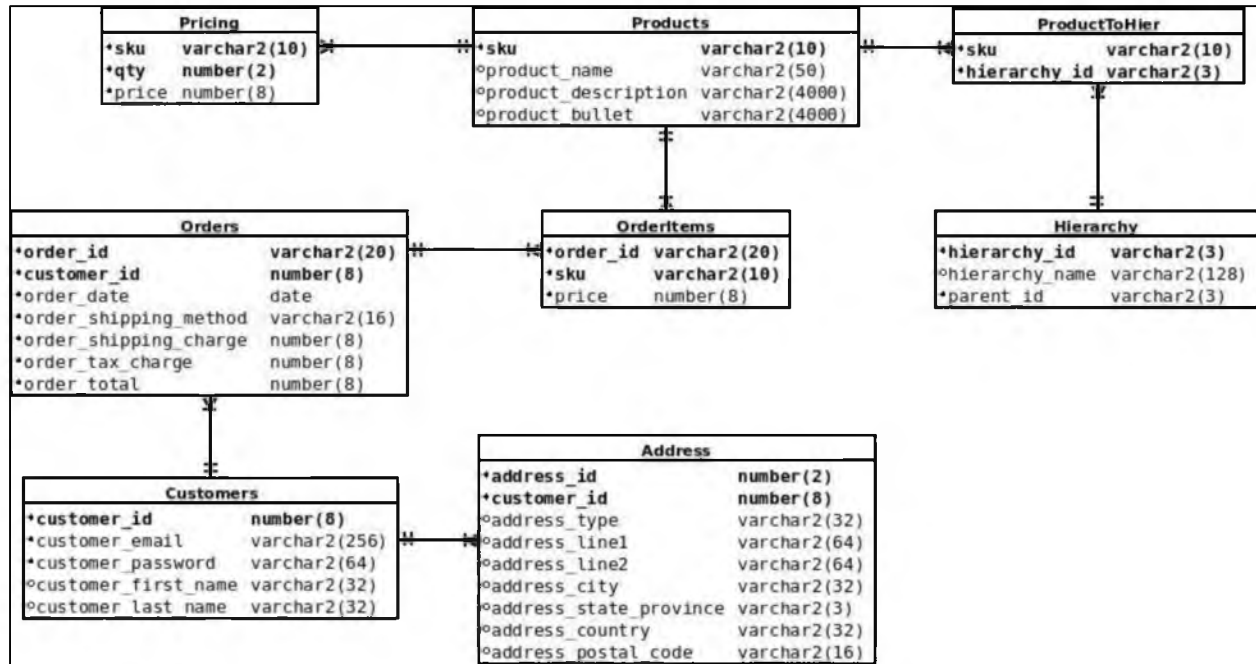
Figure 3-3. Oracle database e-commerce schema (ER diagram).

For Configuration 2 (MongoDB) a de-normalized, non-relational schema was developed (see

Figure 3-4), which employed the use of some redundant data to avoid having to reference it. This

follows the NoSQL practice of (Chodorow, 2011a, ch. 1) "Duplicate data for speed, reference data for

integrity." As mentioned in the previous paragraphs, the use of MongoDB's embedded document

feature was leveraged as well, and this prevented the need for additional data queries. Additional

technical details on the database schema used for the MongoDB experiments can be found in Appendix

E "EcommerceDB Schema (MongoDB)."

```
orders                          customers                          products
{                               {                                  {
        _id,                            _id,                               _id,
        customer_id,                    email,                             sku,
        order_date,                     password,                          product_name,
        order_ship_method,              firstname,                         product_description,
        order_ship_charge,              lastname,                          product_bullet,
        order_tax_charge,               addresses: [                       category_name,
        order_total,                        (                              hierarchy_name,
        order_items: [                          _id,                       pricing: [
            (                                   type,                          (
                _id,                            line1,                             qty,
                sku,                            line2,                             price
                qty,                            city,                          )
                price                           state_province,            ]
            )                                   postal,                    }
        ]                                       country
}                                           )
                                        ]
                                    }
```
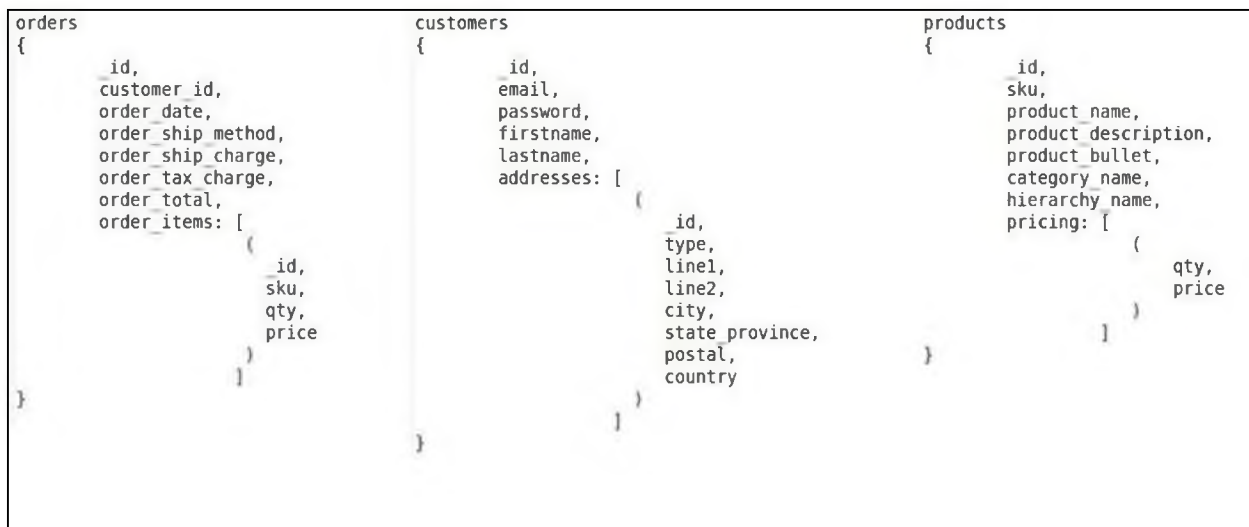
Figure 3-4. MongoDB database e-commerce schema.

De-normalization of the data can also improve disk read times. In a presentation at the 2012 MongoDB Chicago conference, 10gen Solution Architect Chad Tindel showed an example of (Tindel, 2012) how data is stored on-disk for both a normalized and de-normalized schema. Figure 3-5 shows how customer orders are stored on disk in a normalized (relational) e-commerce database.
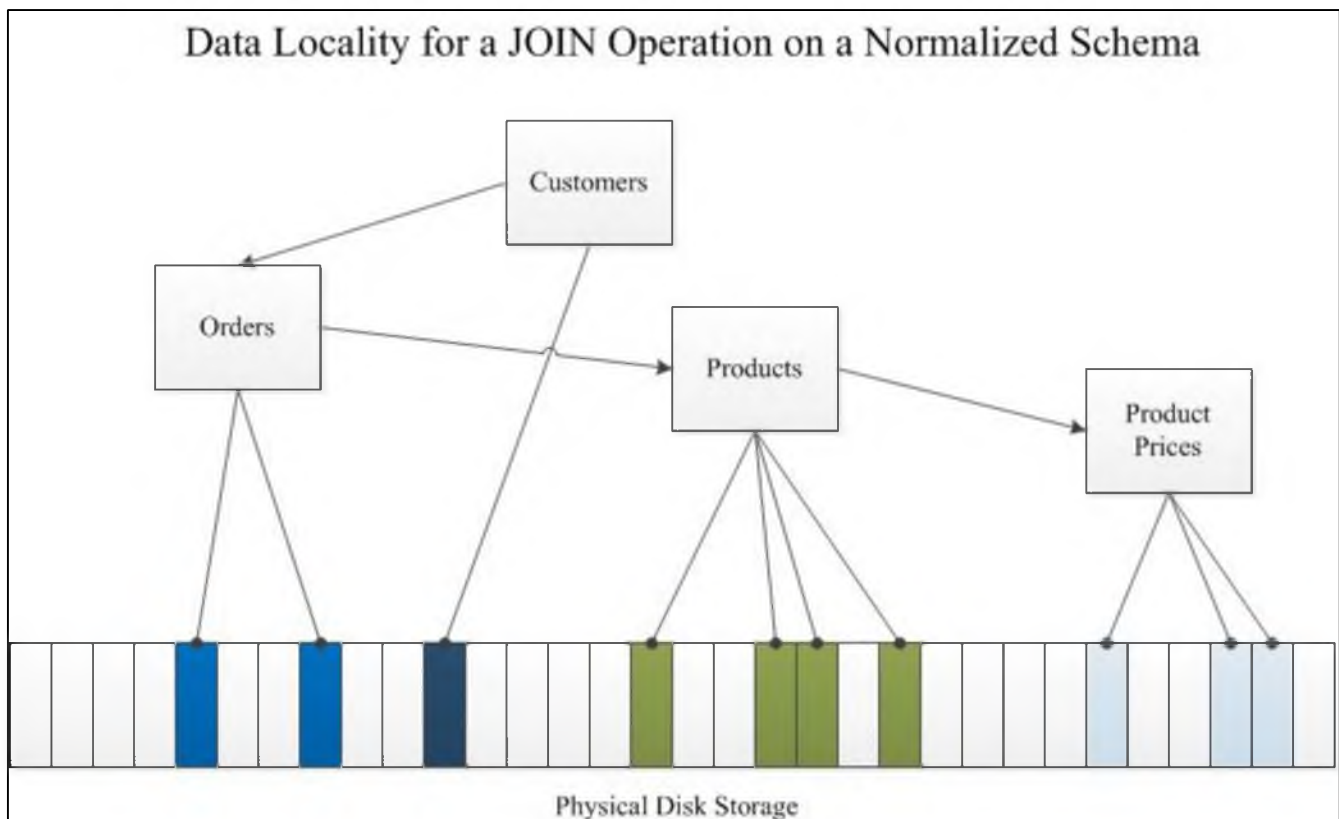
Figure 3-5. Disk storage example for a normalized web order.

To retrieve all of the data for a customer's order, the application or user would need to perform a join operation. Disk seek times are affected because the data must be read from multiple locations on the physical disk, as the parts of the order record are stored in different tables. However, storing customer orders in a de-normalized schema ensures that each order document would contain all necessary information (order data, products, prices, etc) stored continuously on disk. Likewise retrieving this data is faster, because it is read all at once from one spot on the physical disk. Storing data in this manner decreases the time required for reading from disk.

One additional note about MongoDB's collection (table) design is that it is known to have a flexible schema. This allows (Membry, et-al, 2010, ch. 3) MongoDB to support documents with different properties (fields) to "co-exist in a single collection." This is useful when storing documents which contain unique properties that may or may not need to exist on other data elements.

Consider a product database with the property of "batterySize," to contain the designation for the type of batteries that the product takes. In a RDBMS, all products will be required to have a "batterySize" field, even if the product does not require batteries. In MongoDB, only products which required batteries would have a "batterySize" property on the document. All other products would be stored in a document which does not contain the "batterySize" property at all.

Data Generation Tools

Once the databases were installed and the schema was designed, a set of data generation tools was written. First, I created several text files containing hundreds of random data entries for first name, last name, email provider, street name, and city/state/zip combinations. Next, the "generateCustomers" method imported this data, and randomly assembled 250,000 customers each with one or two addresses. This list of customers was then exported to a file, allowing identical customers to be imported into both MongoDB and Oracle.

A similar approach was taken with the product data. I created text files containing several rows of data for product categories, search keywords, and manufacturers. A format for the SKU (product number) was determined to take the two-character category abbreviation and follow it with four digits. This would allow for up to 9,999 products to be randomly created for each category type. The "generateProductData" method then randomly assembled 80,000 products, and exported them to a file. As with the customer data, the same product list was then loaded into both MongoDB and Oracle, so that each would be tested with the same product offering. Technical details on the code used for data generation can be found in Appendix F "Data Generation and Insert Code."

Transaction Benchmarking

I (briefly) explored the use of existing benchmark tools for both MongoDB and Oracle, but could not find one tool that handled both databases. Using one tool for MongoDB and another for Oracle (no matter how similar) would have been using two different measurement strategies, which would have been inherently inaccurate. To ensure accurate benchmarking, the decision was made to set a timer variable (in Java code) just prior to executing a transaction on the database, and to set another timer variable when control was returned to the testing framework. Measuring database activity from within common Java code (Figures 3-7 and 3-8) did offer a consistent way in which both the databases were called and application-level transactions could be measured. The testing framework was coded to function as closely as possible for MongoDB (MongoDB Java driver) and Oracle (JDBC), with the only real difference being the database that was called.

```
String strSQL = "SELECT * FROM products WHERE sku = ? ";

PreparedStatement stmt = conn.prepareStatement(strSQL);

stmt.setString(1, strProductID);

// set start time

long beginDate = System.nanoTime();

// execute transaction

ResultSet rset = stmt.executeQuery();

// control returned to application, set end time

long endDate = System.nanoTime();
```

Figure 3-6. Code sample which measures an application-level transaction for Oracle.

```
DBObject prodQuery = new BasicDBObject();

prodQuery.put("sku", strProductID);

// set start time

long beginDate = System.nanoTime();

// execute transaction

DBObject product = productCollection.findOne(prodQuery);

// control returned to application, set end time

long endDate = System.nanoTime();
```

Figure 3-7. Code sample which measures an application-level transaction for MongoDB.

Preliminary testing indicated that measurement of transactions in milliseconds was not nearly as granular as was necessary for this study. An overwhelming number of transactions yielded a total time of a single millisecond, increasing the difficulty in recording useful data. I then experimented with timing the transactions in nanoseconds, which provided a satisfactory amount of data granularity. Additional technical details on the code used for the testing framework can be found in Appendix G "Testing Framework Code."

It is important to note that there are multiple variables which can affect performance. These variables come from a variety of realms, and can include (but are not limited to) hard disk RPMs, CPU

speed, database indexes, and additional application overhead such as various Java Virtual Machine (JVM) settings. While an effort was made to take these variables into consideration, it is not realistic to test with every possible configuration.

Use Cases

There are several key components of an e-commerce website. An e-commerce site is assumed to be selling something, and their customers must have (Moradi, 2011) quick and intuitive ways to find the products they want. Once customers find their products, (Song, Whang, 2000) the customers have to be able to place orders for them. There also has to be a mechanism to track customer data, and to allow the customers to update their own data. Finally, web merchandisers will need to occasionally update some aspects of the product data.

Based-on the description above, the following use cases were built into the testing framework:

1. Product Navigation

2. Product Searching

3. Order Placement

4. Customer Data Updates

5. Product Data Updates

The testing framework was designed to simulate each of these use cases. This allowed for the databases to be tested while attempting to complete operations that are similar to actual e-commerce based transactions. These transactions (and the results of the testing) will be detailed in the next chapter.

Summary

This chapter was written to accomplish two goals. The first was to discuss the methods of research and design considerations of the testing software, database schema and queries. This will be helpful in understanding the results of the tests on each configuration. The secondary goal was to illustrate the challenges involved in the development phase of this project.

Chapter 4 - Performance Results and Evaluation

This chapter will describe each of the use case-based experiments (for both MongoDB and Oracle), and the results that were returned. The results will be analyzed and compared between each database, and briefly discussed. In an effort to facilitate as direct of a comparison as possible, the same CSV files were used to produce both the customer and product data sets on both MongoDB and Oracle. This helped to ensure that the content of each data row (and the order in relation to the surrounding rows) was identical between each database.

The number of application-level transactions varied by experiment. Experiments based on the customer entity were run once for each customer on-file, or 250,000 iterations. The order-based experiment produced seemingly-random amounts of data, as the number of products purchased (for each order) was randomly generated at runtime. Some of the experiments were not dependent on the number of rows stored, and were run for either 1,000 or 10,000 iterations. These experiments were more dependent on the time for the overall experiment to complete, or the size of the data files produced.

Note that source code and queries for these experiments are not shown in this chapter. This decision was made due to space constraints, as well as not wanting to divert focus from the results or their interpretation. As mentioned in the previous chapter, code for the relevant pieces of the testing framework (including SQL queries) can be found in Appendix G: Testing Framework Code.

Product Navigation Results

Product navigation is a method used by e-commerce customers to locate products for purchase. Typically, the customers have some idea of what they are looking for. The customer then selects a top-level category, followed by one or more sub-categories. On some e-commerce sites, a customer can choose various product attributes to further refine their navigation.

For this experiment, this works in a sequence of three database queries:

1. Top tier product navigation. Customers select their top-level category, and the database

    returns the list of top tier categories for the next choice.

2. Product group navigation. The customer selects a category, and the database returns the list

    of products under that category.

3. Single product navigation. The customer selects a product, and the database returns all of

    the data associated with that product.

To test this use case, three queries were run for 1,000 product navigation transactions each,

against both databases:

1. Query #1 (retrieval of the top tier categories) on Oracle (Figure 4-1) took an average of

    53.797ms, while running in an average of 153.332ms on MongoDB (Figure 4-2).
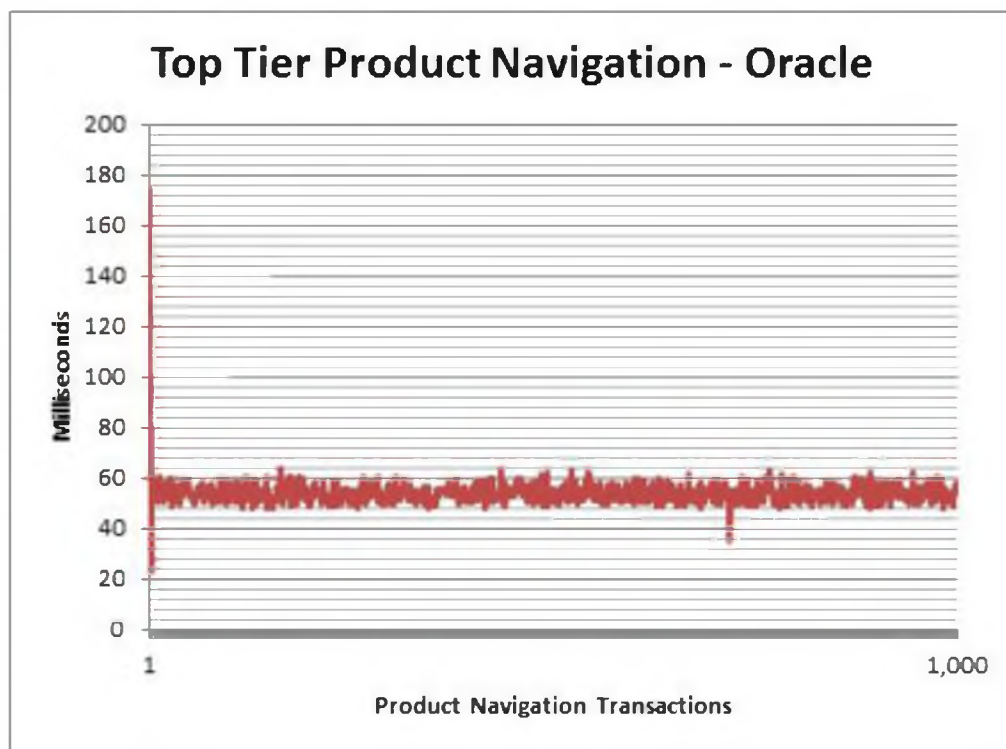
Figure 4-1. Performance statistics for retrieval of top tier product categories from Oracle.
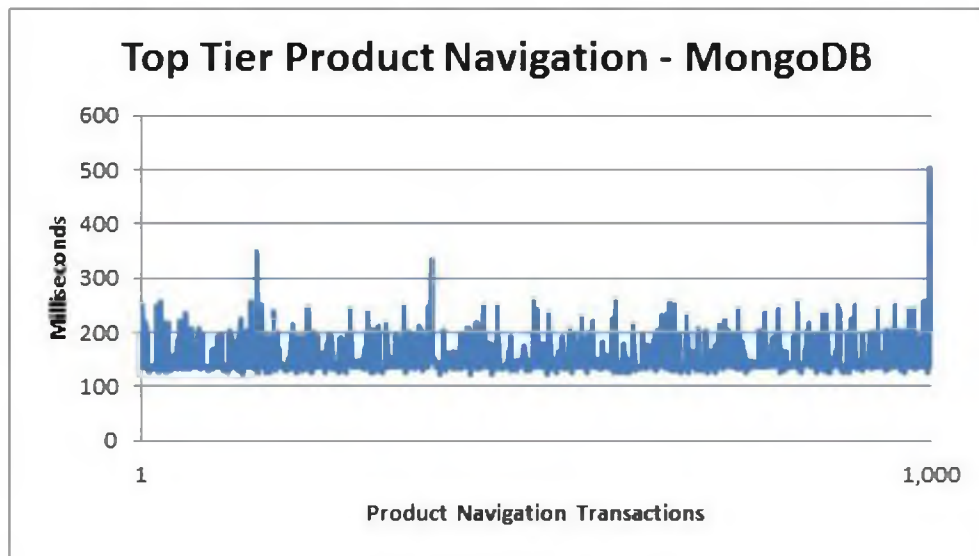
Figure 4-2. Performance statistics for retrieval of top tier product categories from MongoDB.

2. Query #2 (retrieval of the product group's product list) on Oracle (Figure 4-3) took an average of 4.411ms, while running in an average of 220.479ms on MongoDB (Figure 4-4).
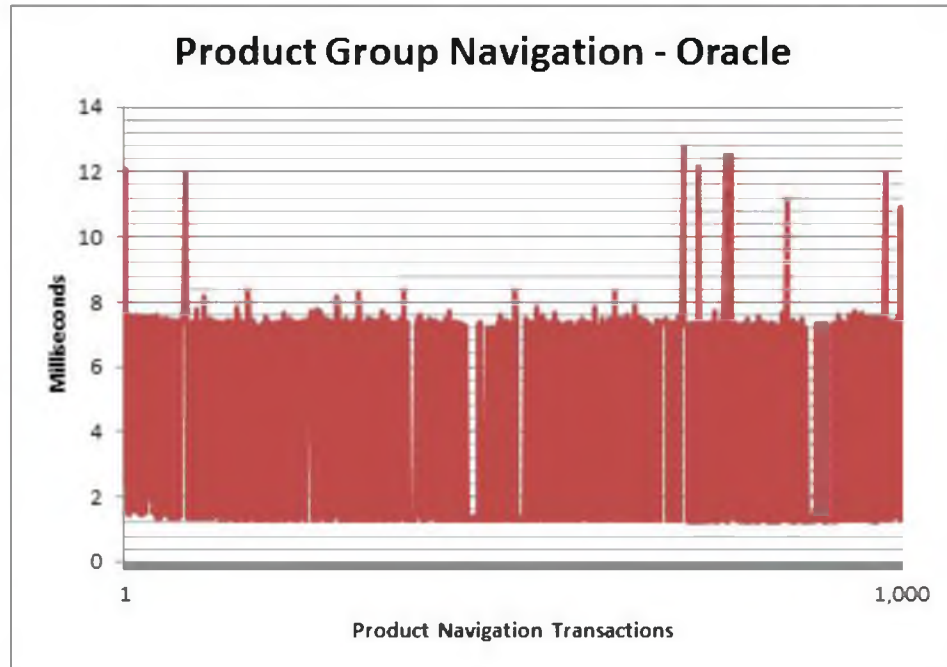
Figure 4-3. Performance statistics for retrieving the product groups from Oracle.
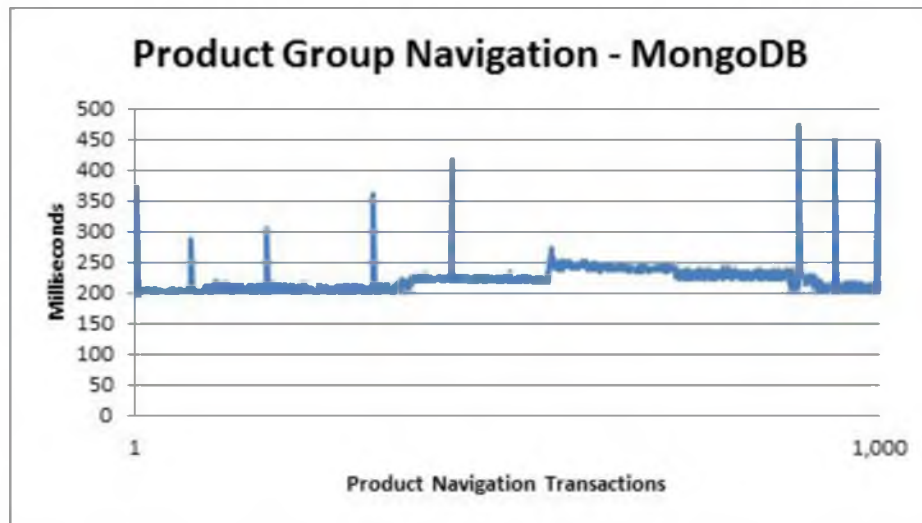
Figure 4-4. Performance statistics for retrieving the product groups from MongoDB.

3. Query #3 (retrieval of the data for a single product) on Oracle (Figure 4-5) took an average of

13.318ms, while running in an average of 1.726ms on MongoDB (Figure 4-6).
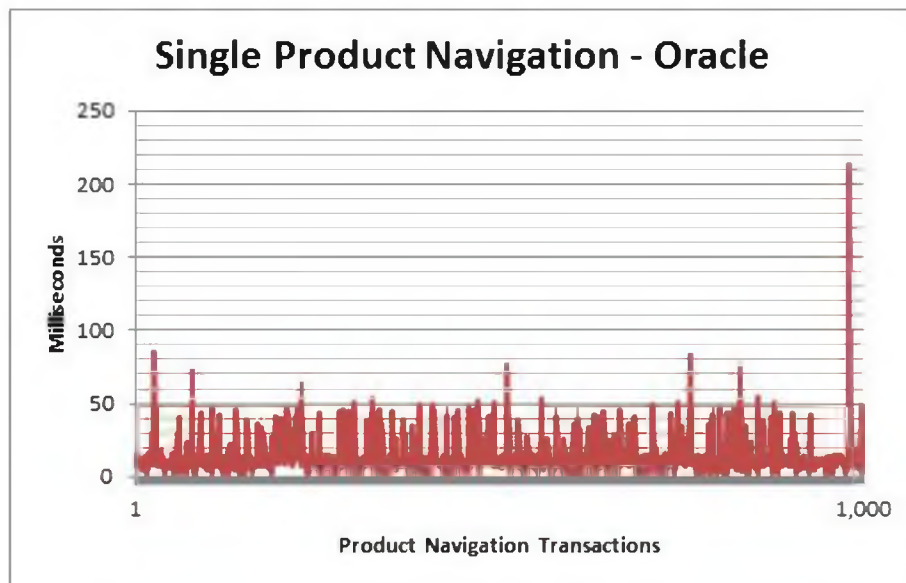
Figure 4-5. Performance statistics for retrieval of a single product form Oracle.
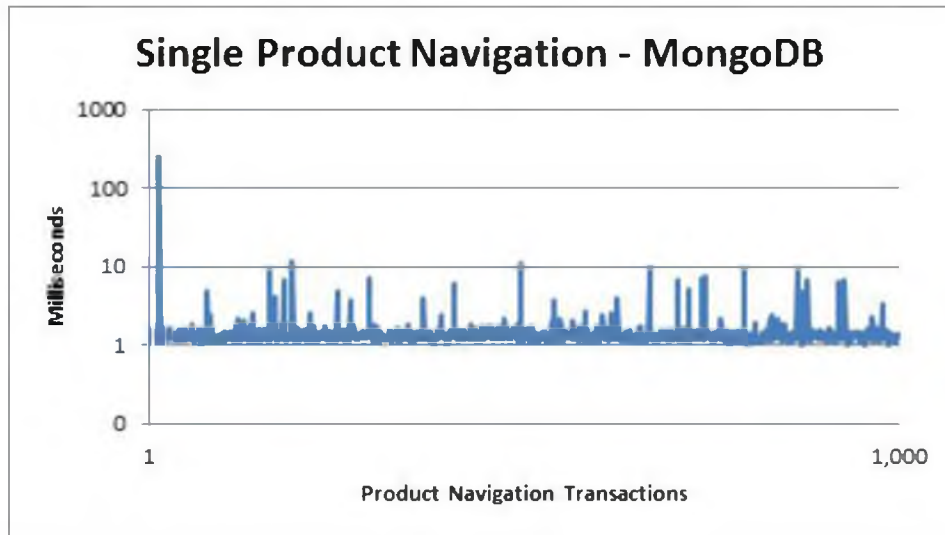
Figure 4-6. Performance statistics for retrieval of a single product from MongoDB.

Product Search Results

Although guided navigation is an excellent feature, every e-commerce website should have a well-functioning keyword-based search. Users who come to a website already knowing what they want need a way to search for it. These users may already have some information on the product (type, specification, product name, etc…), which should allow them to locate it.

Neither table utilized an index on the product_description field, as the product_description is usually a "payload field." The field's maximum size of 4000 characters allows it to contain free-form text, which could hamper attempts to quickly locate data. The product_description field was selected for this experiment because it is a subpar choice. However, the performance results of the regular expression transactions could be more easily examined.

This experiment was driven by a list of 17 search terms. The terms were compiled from a file of (fictitious) companies and a file of product description keywords. Both files were used in the random generation of the product data.

This experiment made use of the regular expression matching functionality offered by both

MongoDB and Oracle. A regular expression search was run against the product description, which was

generated as a random composition of keywords, company names and noise words. Further study

seemed to indicate that the difference between case sensitive and case insensitive matching was

significant to performance, so transactions were run for each. A series of 10,000 transactions was then

executed against each database, and yielded the following results:

1. A case sensitive regular expression query against the products table ran in an average of

    366.5300ms for Oracle (Figure 4-7), while running in an average of 0.004491ms (4491ns)

    for MongoDB (Figure 4-8).

Figure 4-7. Performance statistics for a case-sensitive keyword search from Oracle.
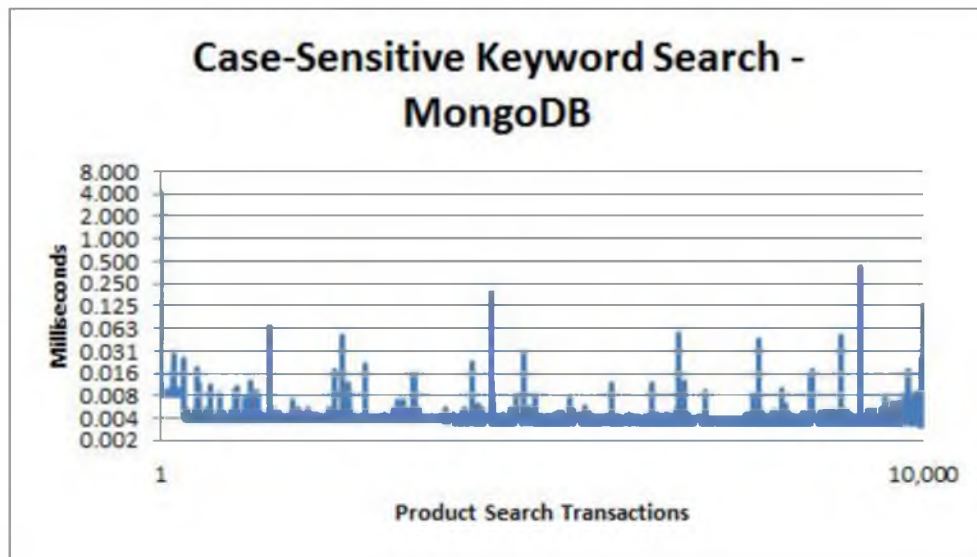
Figure 4-8. Performance statistics for a case-sensitive keyword search from MongoDB.

2.   A case insensitive regular expression query against the products table ran in an average of

194.4144ms for Oracle (Figure 4-9), while running in an average of 0.004937ms (4937ns)

for MongoDB (Figure 4-10).

Figure 4-9. Performance statistics for a case-insensitive keyword search from Oracle.
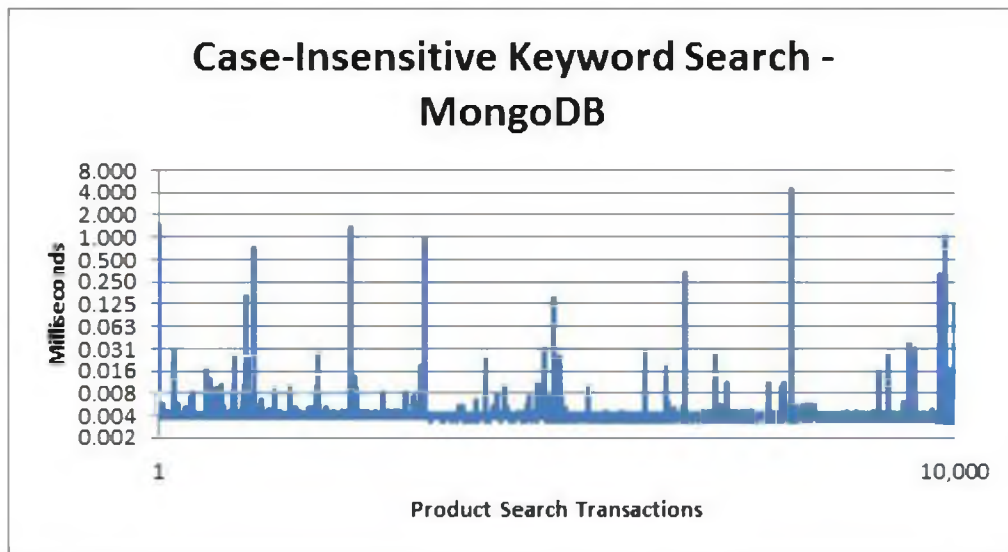
Figure 4-10. Performance statistics for a case-insensitive keyword search from MongoDB.

Customer Update Results

This use case was designed around a customer needing to update his/her website account data. A common e-commerce user task is the updating of the account password. This specific type of update is also easily tested.

It should be noted that this experiment did not adhere to PCI (Payment Card Industry) standards concerning the storage of passwords in plain-text. The topic of PCI compliance is not in the scope of this project or these experiments. However, anyone building an e-commerce website should adhere to the PCI stipulation (PCI, 2010, p. 19) to "Render all passwords unreadable during storage and transmission, for all system components, by using strong cryptography."

This experiment was designed to read a text file containing each of the 250,000 email addresses in the customer table/collection. The same file was run against both MongoDB and Oracle, allowing each database to be tested using update queries that ran with the same addresses in the same order. As the testing framework iterated through the email addresses in the file, it also generated a "strong" eight-character password. With the email and password in-hand, the framework sends a single update transaction to the database, simulating an update of the user's password. The run of 250,000

transactions for each database yielded the following results:

1. A simple update against the customers table ran in an average of 0.8465ms for Oracle (Figure 4-11, while running in an average of 0.2841ms for MongoDB (Figure 4-12).
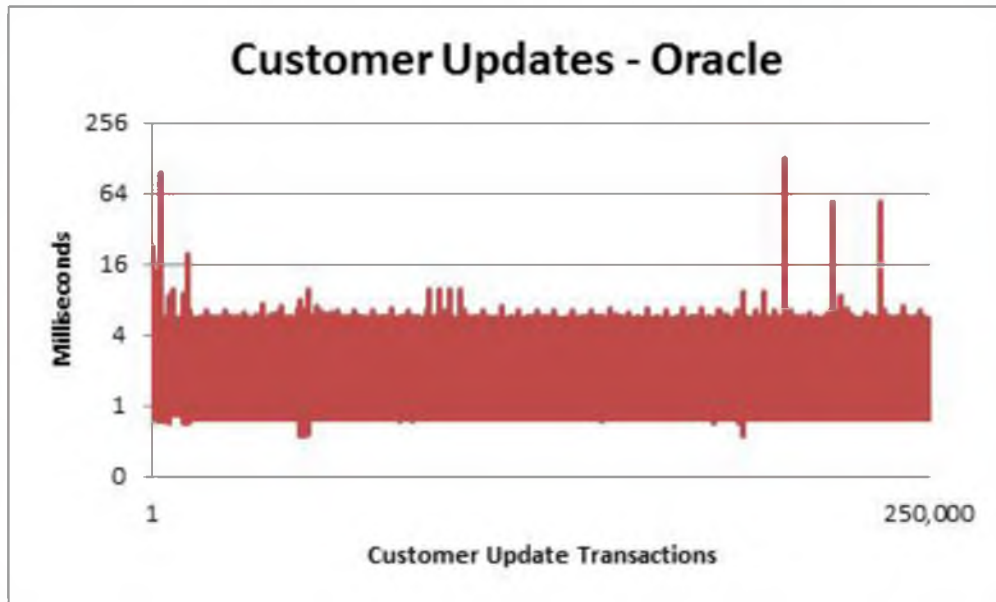
Figure 4-11. Performance statistics for customer updates on Oracle.
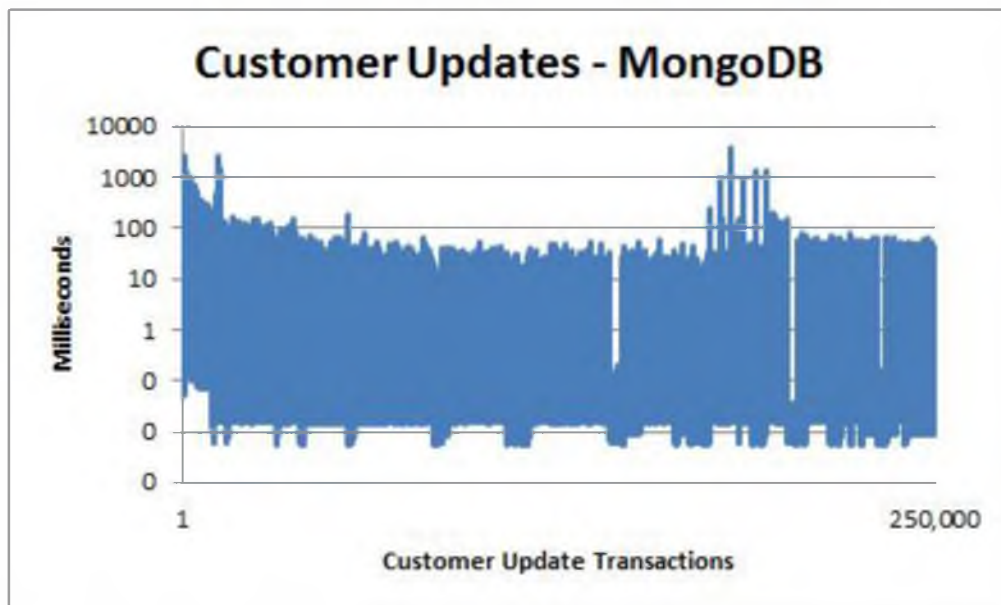
Figure 4-12. Performance statistics for customer updates on MongoDB.

Order Placement Results

By their very nature, all e-commerce websites need to be able to store and track orders placed by customers. A customer order contains a list of products (and quantities) to be ordered. It also contains information about the customer, such as shipping address and payment information.

It should be noted that this experiment did not adhere to Payment Card Industry (PCI) standards referring (PCI, 2010) to the transmission and storage of unencrypted payment information. The topic of PCI compliance is not in the scope of this project or these experiments. However anyone building an e-commerce website should adhere to the PCI stipulation of never storing payment information unless (PCI, 2010, p.14) it is "necessary to meet the needs of the business." Regardless, payment information should also be encrypted as soon as it enters the application.

For this experiment, 10,000 orders were placed against both databases by randomly-assigned customers. Only the customer's ID was attached to the order. This way, the customer data could be available for reference on an as-needed basis. Each order was then given between one and five (random) products, each with a quantity between one and twelve (random). This resulted in a total product count of 29,856 (or 2.9856 products per order) for Oracle and 30,088 products (3.0088 products per order) for MongoDB.

The product prices were then queried from the database, so that the correct quantity-break pricing could be established. Once the product prices were obtained, the order data was then inserted into the "orders" table/collection.

One difference in the ordering process between Oracle and MongoDB, was in how the ordered products were stored in the databases. In Oracle, the products were inserted to the "order_items" table

in an additional transaction that took place after the initial insert to the "order" table. In MongoDB, the products were written into an embedded array on the order document, and inserted into the "order" collection along with the rest of the order data.

This resulted in the following database transactions:

1. A pricing query against the "products" table/collection ran in an average of 1.841ms for Oracle (Figure 4-13), while running in an average of 0.005401ms (5401ns) for MongoDB (Figure 4-14).
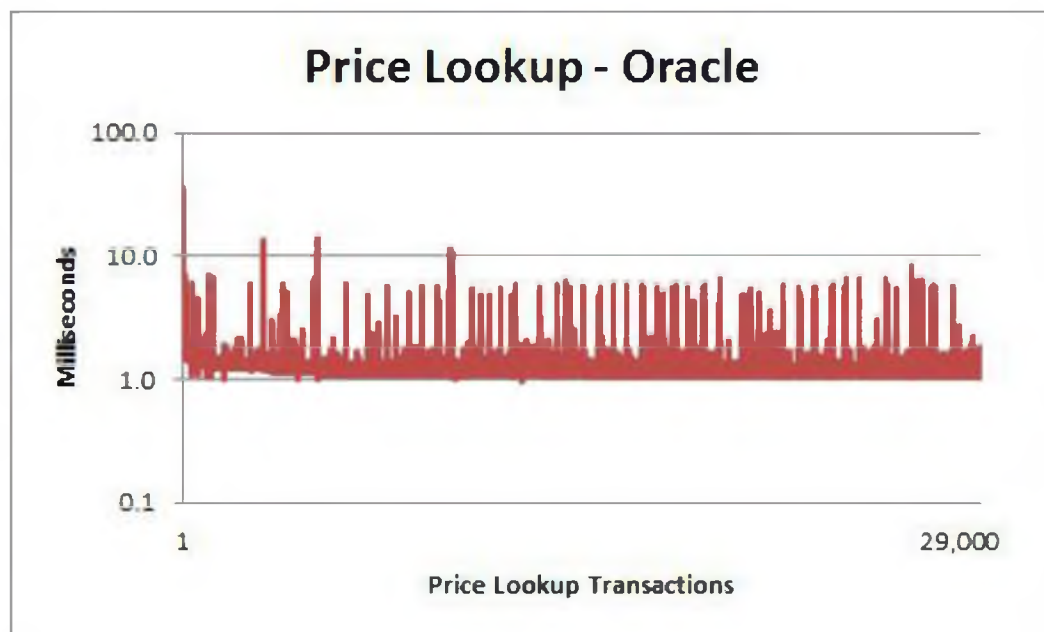
Figure 4-13. Performance statistics for a price lookup from Oracle.
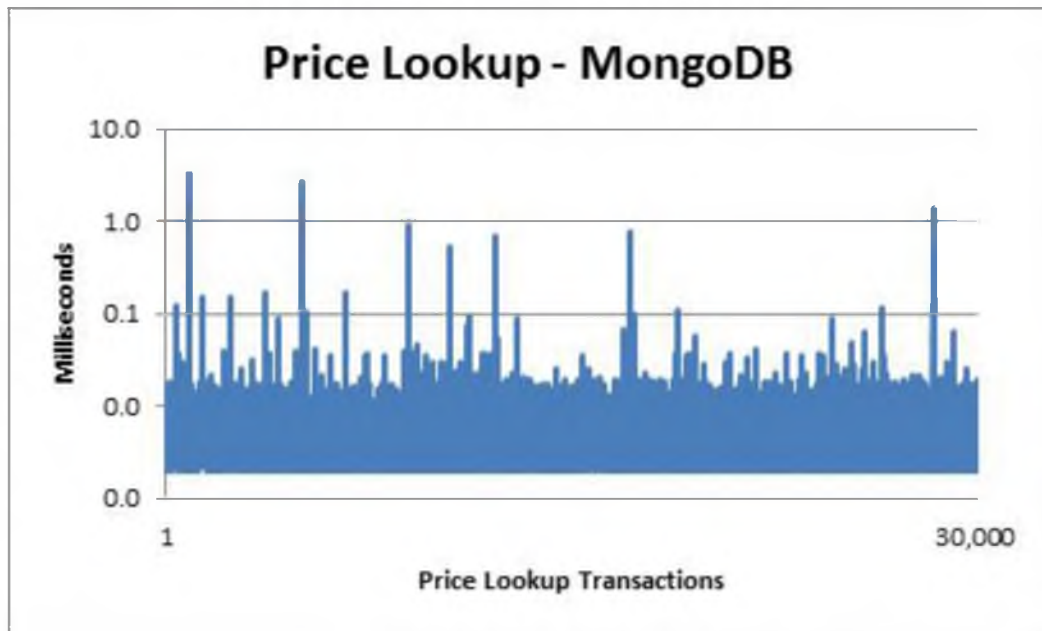
Figure 4-14. Performance statistics for a price lookup from MongoDB.

2. An insert into the "orders" table/collection ran in an average of 20.312ms for Oracle (Figure 4-15), while running in an average of 0.1179ms for MongoDB (Figure 4-16).
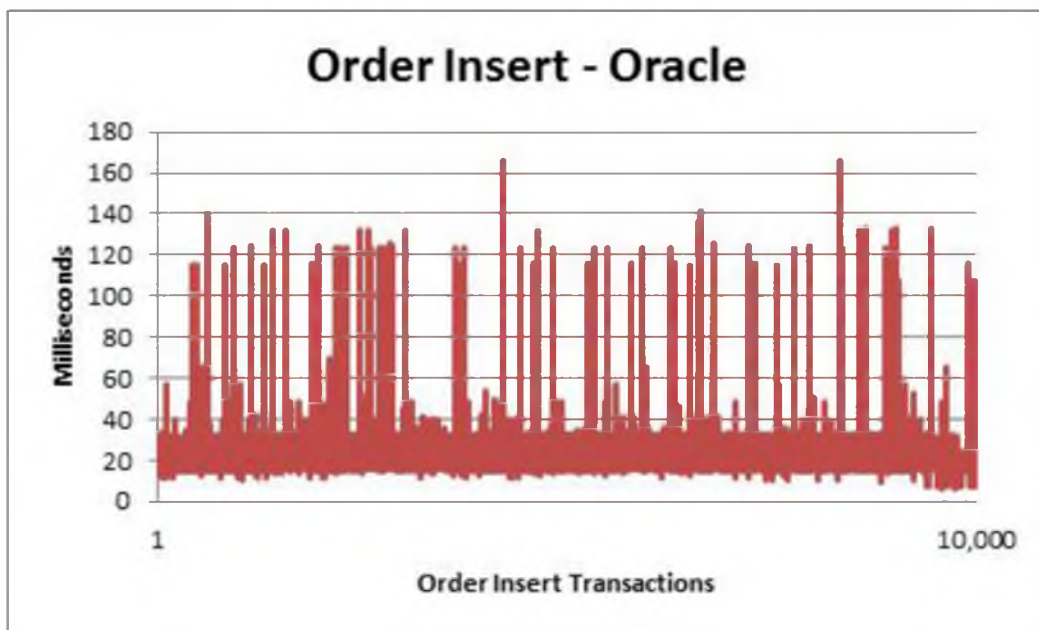
Figure 4-15. Performance statistics for an order insert to Oracle.
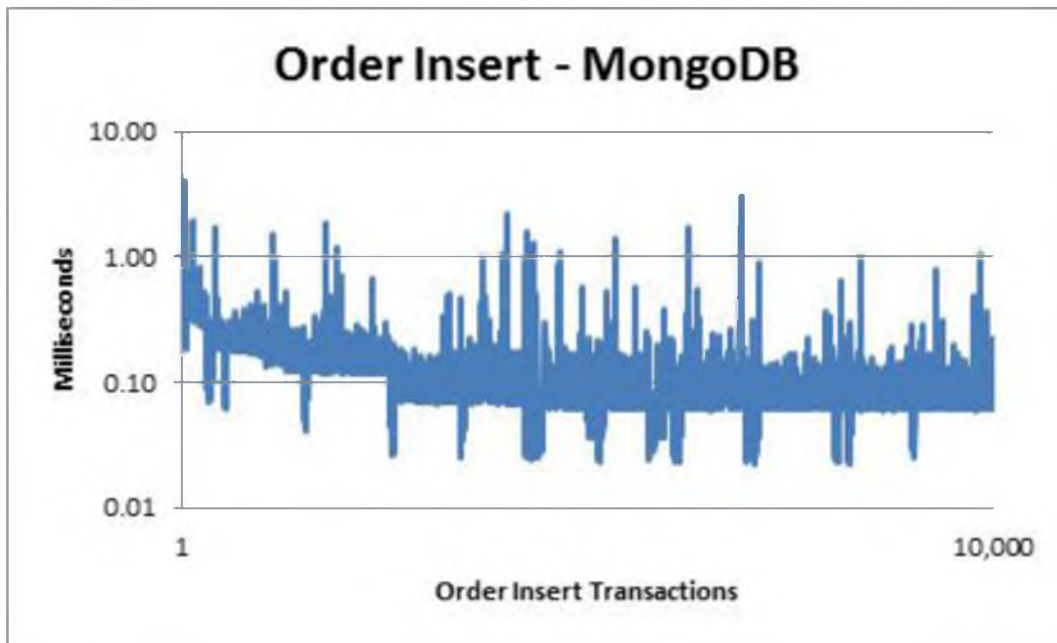
Figure 4-16. Performance statistics for an order insert to MongoDB.

3. An additional insert into the "order_items" table was required only for Oracle (Figure 4-17), and ran in an average of 20.513ms.
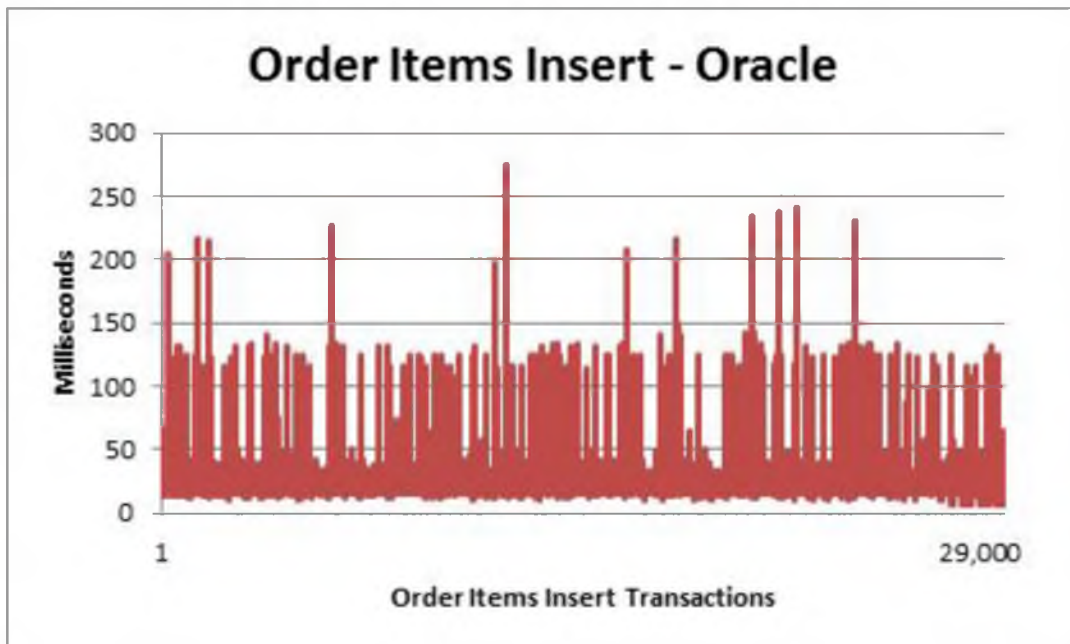
Figure 4-17. Performance statistics for an order items insert into Oracle.

Product Price Update Results

In an effort to drive sales, web merchandizers occasionally offer certain products at discounts, in bulk or in combination with other products. This fluctuating nature of product prices makes them a good topic for product update experimentation. In e-commerce, many product prices are subject to change based on the quantity ordered (known as a quantity break), and the data for this experiment was designed to reflect this.

This experiment simulates a product manager performing an alteration to a price for a specific product and quantity break. To provide us with an adequate data set, 10,000 product price update transactions were executed. This resulted in the following database transactions:

1. A pricing update against the "products" collection and "pricing" table ran in an average of 19.65ms for Oracle (Figure 4-18), while running in an average of 0.442ms for MongoDB (Figure 4-19).

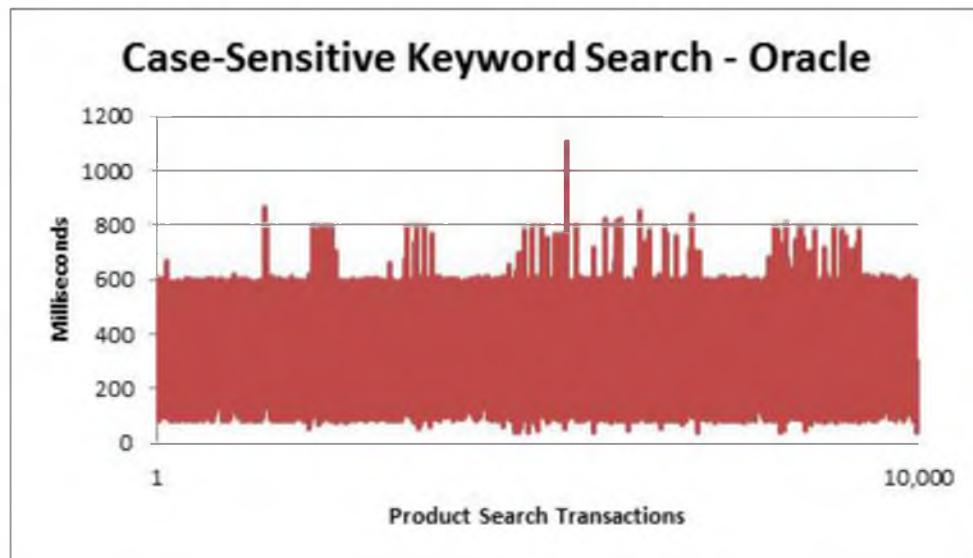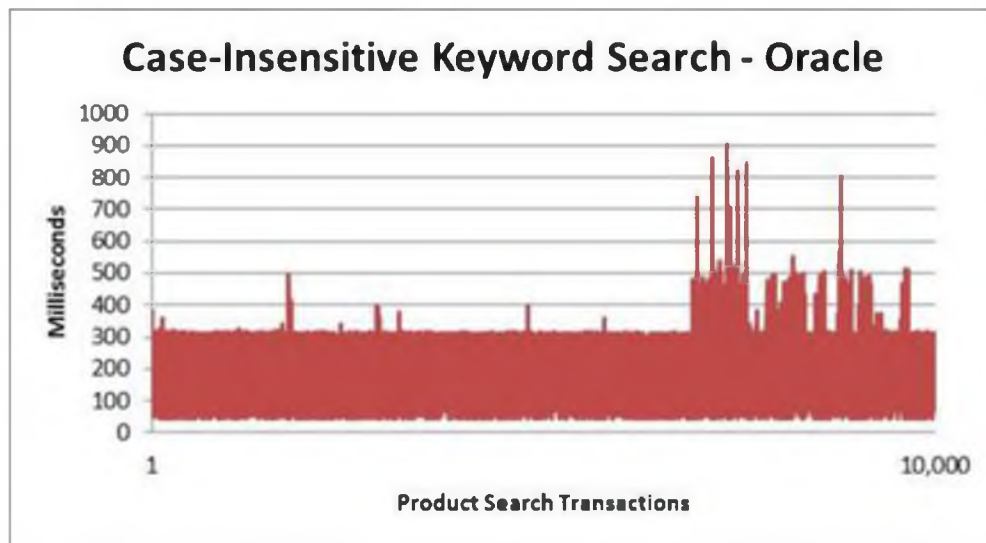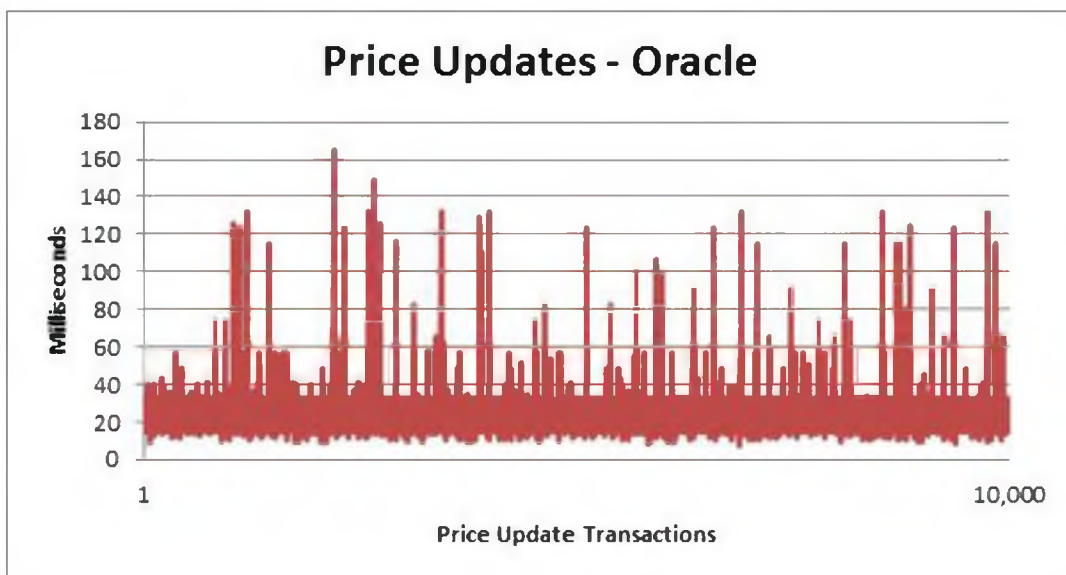Figure 4-18. Performance statistics for price updates in Oracle.
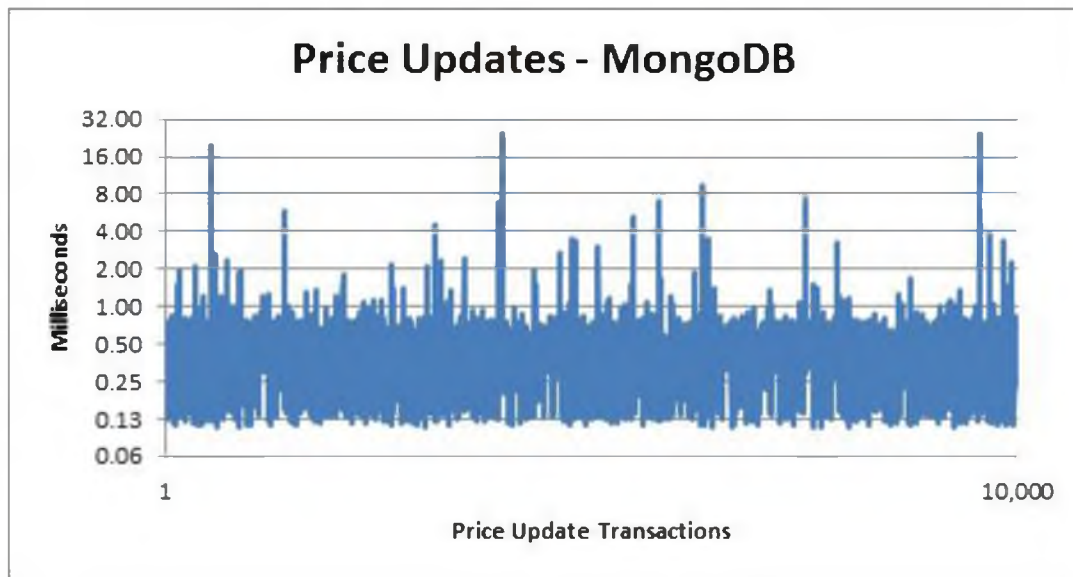
Figure 4-19. Performance statistics for price updates in MongoDB.

Product Navigation Data Interpretation

The results of queries #1 (Top Tier Navigation) and #2 (Product Group Navigation) were superior while running on Oracle. Additionally, the corresponding queries running on MongoDB exhibited sub-par performance. In the specific case of query #2, Oracle was running more than 50 times faster than MongoDB.

Query #3 (Single Product Navigation) however, performed as expected. This particular query ran more than 11 times faster on MongoDB than it did on Oracle. This would suggest that returning full documents from MongoDB was much faster when using a query that was premised on uniqueness (returning only one result).

The schema was examined to see if any improvements could be made. Although NoSQL databases (MongoDB, specifically) are known for achieving performance through de-normalization, I theorized that the hierarchy-to-category-to-SKU navigation queries might perform faster if the schema was re-engineered to produce a smaller amount of raw data. Or rather, that the queries concerned with

the hierarchy, category, and category-to-SKU relationships would perform faster if they were not

"weighted-down" with extra data like product_description, product_name, and pricing.

To test this theory, a collection named "categories" was created containing the hierarchy_name,

category_name, and SKU fields. This allowed the category_name (renamed to "category" in

MongoDB) field to be made unique. The redundant hierarchy_name fields were added to the category

documents (as "hierarchy"). The category-to-SKU relationship was also maintained here, as each

category document was given its list of related SKUs in the form of an embedded array.

After a re-run of 1,000 transactions (only for queries #1 and #2), it was clear that this change

had achieved a superior result. The MongoDB transactions for queries #1 and #2 had become several

orders of magnitude faster:

1. Top tier product navigation queries now ran in an average of 0.6701ms for MongoDB

    (Figure 4-20), which is much faster than the original average of 153.332ms (Figure 4-2).
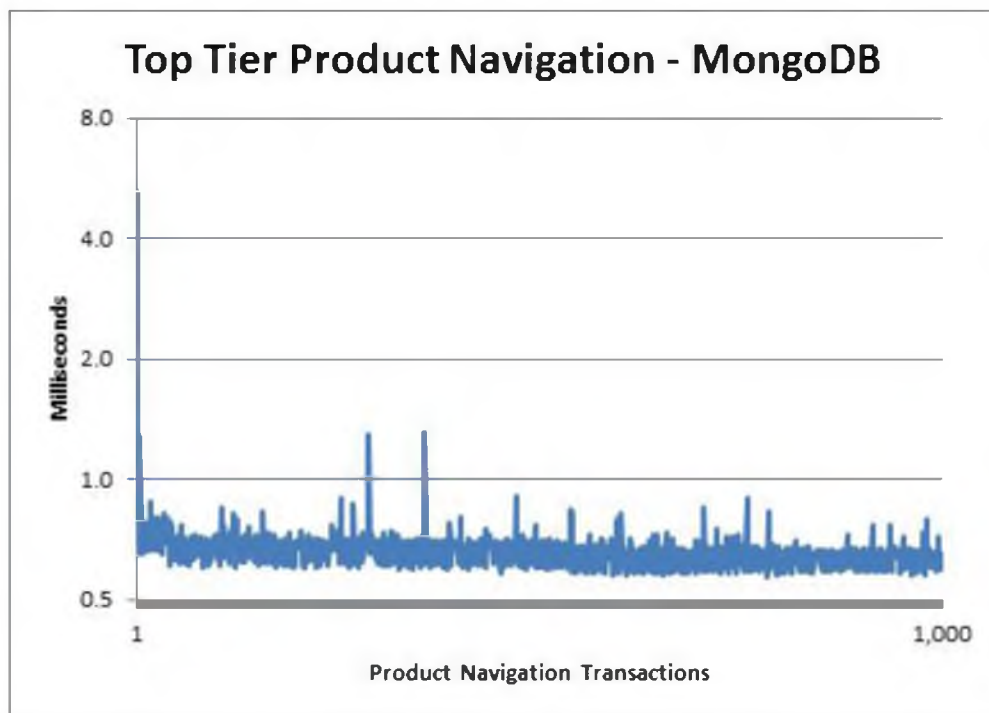
Figure 4-20. Post schema change performance statistics for retrieval of top tier product

categories from MongoDB.

2. Product group navigation queries now ran in an average of 0.0085ms for MongoDB (Figure

4-21), which is much faster than the original average of 220.479ms (Figure 4-4).
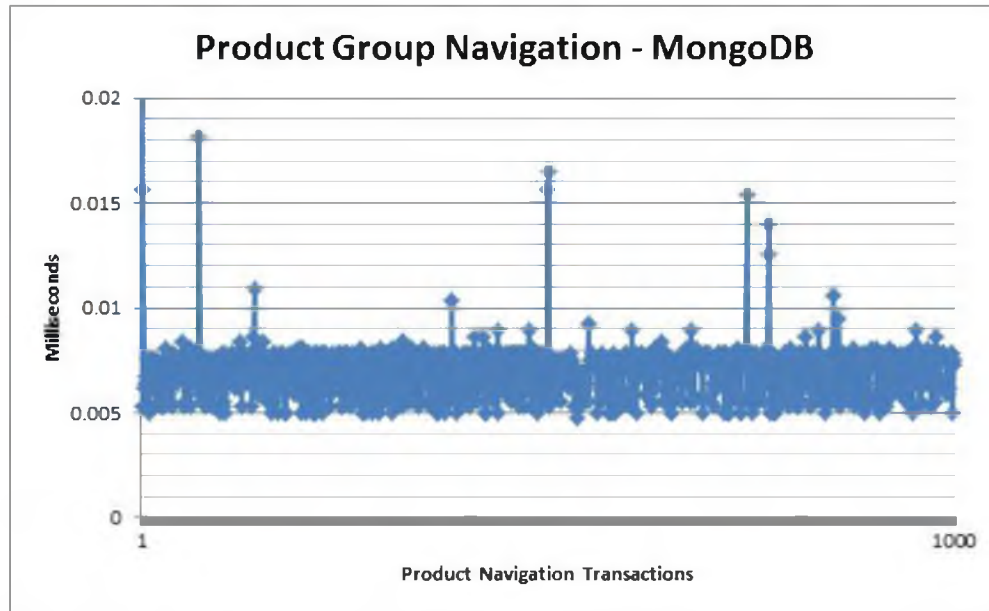
Figure 4-21. Post schema change performance statistics for retrieving the product groups
from MongoDB.

The biggest aspect emphasized by this particular experiment, is that schema design in

MongoDB (or any database, for that matter) is key to building a well-performing database. As stated

by author Hennie Grobler in his 2011 paper "Using MongoDB for an E-Commerce Platform,"

(Grobler, 2011, p. 10) "One of the biggest factors in deciding how the data is modeled depends on how

the entities are accessed in relation to one another." Grobler goes on to say that if certain entities are

usually accessed individually, then they should be separated from each other. This makes sense, as use

cases calling for hierarchy and category data would seldom require product details.

Product Search Data Interpretation

The most prominent result in this experiment is that the MongoDB regular expression searches

completed much faster than their Oracle counterparts. This was especially true of the test involving

case sensitivity. I investigated this phenomena and discovered that MongoDB and Oracle use different regular expression libraries.

Oracle introduced regular expressions with Oracle 10g (Casteel, 2007, p. 328), which uses an implementation of the POSIX IEEE regular expression library. MongoDB regular expressions (Banker, 2012) utilize the Perl-Compatible Regular Expressions (PCRE) library. The most notable direct comparisons between the two can be examined via the PHP language, which has utilized both libraries in the past, having deprecated the POSIX library as of (PHP) version 5.3. One obvious inference from this experiment would be that the PCRE regular expression library performs faster than the POSIX library. However, finding empirical comparisons to support this inference proved to be challenging.

One such source was an addendum to the PHP Manual by Pedro Freire, PhD. Freire posted the results of a variety of regular expression tests (Freire, 2007) that he had run (using PHP 5.1.2 on Windows) to compare the PCRE and POSIX regular expression libraries for PHP. While Freire listed results that favored PCRE in most instances, there are a few complex matching problems that the POSIX library was able to resolve faster.

In his experiment on case-insensitive string pattern matching, Freire reported (Freire, 2007) that his PCRE test performed 30 times faster than its POSIX counterpart. He also tested transactions where the case sensitivity was specified in the expression itself (not invoked through the POSIX library), but achieved results that were inconsistent. Freire labeled his own results as inconclusive and recommended that developers should (Freire, 2007, para. 22) "always benchmark your PCRE / POSIX RE to find the fastest!"

Another useful source was a comparison from Boost. While Boost was promoting its own regular expression libraries (intended to be used in C++), their comparisons (Maddock, 2003) included

benchmarks from both POSIX and PCRE library calls, which heavily favored PCRE. Author and developer John Maddock, PhD, made sure to mention that "care should be taken in interpreting the results, only sensible regular expressions (rather than pathological cases) are given."

It is important to note that any conclusions inferred from these sources (and this experiment) are assuming similarities in the POSIX and PCRE library implementations between PHP, C++, MongoDB and Oracle. Without further study on this particular topic, there is little certainty on how much similarity may or may not exist. As for the results in this experiment, the MongoDB transactions ran in 4491ns with case-sensitivity enabled and 4937ns as case-insensitive. The difference of 441 nanoseconds is negligible.

Conversely, Oracle came in at 366.53ms on case sensitive searches and 194.4144ms on case insensitive searches. These two results illustrate a significant difference and suggest that the Oracle/POSIX implementation is more adept at handling case insensitivity. I would have expected to find the opposite to be true here, as invoking case insensitivity essentially widens the spectrum of possible matches and likewise increases the potential size of the data set returned.

It would appear that the regular expression library implementation (used in each database) may have an influence on performance. The exact reasons behind it cannot be easily ascertained, and are also beyond the scope of this project. Suffice to say that in this particular instance, executing 10,000 string pattern matches (regardless of specification of case sensitivity) on identical datasets performs significantly faster with MongoDB/PCRE than Oracle/POSIX.

Customer Update Data Interpretation

In this experiment, both databases performed well. While the transaction running against MongoDB performed almost three times faster than its Oracle counterpart, both are in the sub-millisecond range. This level of performance for either database should be more than sufficient for e-

commerce customer updates.

Order Placement Data Interpretation

Both databases performed well during the pricing query. MongoDB did perform this query much faster. However, Oracle was still averaging in the sub-two millisecond range, which is still quite fast.

The inserts transaction speeds posted were in MongoDB's favor, with MongoDB performing the inserts more than 170 times faster than the corresponding Oracle inserts. It should be noted that Oracle did have an additional insert transaction (due to normalized schema constraints). However both inserts exhibited similar performance (20.312ms for "orders" and 20.513ms for "order_items").

Product Price Update Data Interpretation

The update transactions for MongoDB completed almost 45 times faster (19.65ms vs. 0.442ms, respectively). However, comparing the numbers from this experiment to the performance statistics for previous experiment on "Order Placement" reveals an interesting observation. Oracle performed slightly faster (between .662ms and .863ms) on an update vs. an insert. MongoDB's update average of .442ms was almost four times slower while performing the update (vs. an insert). While it makes sense that an update should be slower than an insert, the astonishing part here is how Oracle's performance remains more-consistent than MongoDB's.

Summary

In summary of the performance experiments, the most prevalent observation is that MongoDB out-performed Oracle in almost every instance. The first two parts of the product navigation experiment where Oracle out-performed MongoDB, was rectified with a slight alteration in the

MongoDB schema. This suggests that although MongoDB (and other NoSQL databases) may advertise themselves as schema-less or as having a (Membry, et-al, 2010, ch. 3) flexible schema, that schema design is still important to the performance of an application.

MongoDB performed well with a regular expression search. Although slower, Oracle seemed to perform faster utilizing a case-insensitive search. Conversely, case-sensitivity did not make a significant difference with MongoDB.

On the insert-based experiments, MongoDB performed much faster than Oracle. Although taking four times as long on its update operations, MongoDB still performed these operations in an average of less than 0.5 milliseconds. This contrasted with the Oracle updates, which performed more-consistently relative to its recorded insert times.

The goal of this chapter was to present the results of the various experiments in a way that allowed their differences to be easily observable. In some cases, further research and experimentation was required, and executed when believed to be within the scope of this project. These results do present additional opportunities for more-focused study into areas such as the regular expression libraries used by MongoDB and Oracle, as well as how/if extra processing required to comply with PCI standards may affect performance.

Chapter 5 - MongoDB ACID Experiments Results and Evaluation

One of the main strengths of RDBMSs (including Oracle 11g R2), is the ability to ensure ACID transactions. This essentially means that transactions are guaranteed to be atomic, consistent, isolated and durable. On the other hand, NoSQL databases are known for (Decandia, Hastorun, Jampani, et-al, 2007) sacrificing certain aspects of ACID to achieve superior levels of performance. Again, this is because Gilbert and Lynch established that (Gilbert, Lynch, 2002, p.11) "it is impossible to reliably provide atomic, consistent data when there are partitions in the network."

MongoDB is no exception in this case. It is classified with a "CP" (strong-consistent and partition-tolerant) CAP configuration, which means that its designers (according to Brewer's CAP Theorem) decided to support partitions over a network and strong-consistency while dropping high-availability. The question to ask is how do MongoDB's transactions behave relative to ACID? This next chapter will evaluate MongoDB in adherence to ACID transaction properties.

It should be noted that this section will not discuss Oracle 11g R2's adherence to ACID nor its "CA" CAP configuration. It is assumed that Oracle 11g R2 is fully ACID compliant. This is designation is a strength of and typically a primary reason for selecting (or staying with) a RDBMS over a NoSQL database.

Atomicity Results

To test MongoDB's adherence to atomicity, I created a new collection called "postings" (in reference to a hypothetical blog application). After querying for a "`count()`" on this collection (via a MongoDB shell), it was clear that the number of documents contained within was zero. I then executed the following JavaScript (shown in Figure 5-1) from the MongoDB of a local MongoDB instance:

```
> $counter=1;

    while(true){

        db.postings.insert({_id:$counter,voter:"test"});

        $counter++;}
```

Figure 5-1. Code to insert an infinite amount of documents into the "postings" collection.

After waiting a few seconds, I terminated the operation. I then queried the "postings" collection

from a MongoDB shell. As shown in Figure 5-2, the "postings" collection now contained 327,951

documents.

```
> db.postings.find().count()
327951
```

Figure 5-2. Query of the number of documents in the "postings" collection.

Next, I applied an update to the above records that made it into the database. Code to update all

records to have a "status" property equal to the string "updated" (shown in Figure 5-3) was run and

quickly killed (with a control-C). I then re-entered the shell and queried the "postings" collection to

see that 94,208 documents now had a "status" property of "updated".

```
> db.postings.update({},{$set:{status:"updated"}},{multi:true})


do you want to kill the current op(s) on the server? (y/n): y
$ ./mongo
MongoDB shell version: 2.2.2
connecting to: test
>   db.postings.find({status:"updated"}).count()
94208
```

Figure 5-3. Shell showing an update to all records, break, and query a count of updated documents.

Consistency Results

As mentioned, MongoDB is a CP (strong-consistent and partition-tolerant) database. In a

partitioned MongoDB environment, writes are made to a master server and replicated to the remaining

nodes. There is typically a delay between a write to the database and that write being persisted across all nodes, leading to a (temporary) compromise of data integrity. The time to achieve data consistency is proportional to the amount of data that is being sent to the replica node.

I ran a quick and simple experiment to demonstrate MongoDB's replica set consistency, and the amount of times taken to achieve a consistent state. Two MongoDB replicas were created on the same machine (note that you would typically create replicas on separate machines). The replicas (10gen 2012) were started-up and configured, allowing one to be voted "PRIMARY" and the other "SECONDARY." Roughly 180 megabytes of test data was generated (similar to how data was generated in the atomicity experiment in the prior section) and inserted into the "PRIMARY" member.

Once this initial set up was completed, a third replica set member was created and started (at 18:13:15 GMT). The status of the new replica set member was queried several times from a MongoDB shell. The output shown in Figure 5-4 indicates that the state of the new replica set member is "DOWN" and that it is currently initializing.

```
"_id" : 3,

"name" : "localhost:27003",

"health" : 1,

"state" : 8,

"stateStr" : "DOWN",

"uptime" : 11,

"optime" : Timestamp(0, 0),

"optimeDate" : ISODate("1970-01-01T00:00:00Z"),

"lastHeartbeat" : ISODate("2013-01-06T18:13:26Z"),

"pingMs" : 0,

"errmsg" : "still initializing"
```

Figure 5-4. Excerpt from the "rs.status" command, showing statistics for the new replica set member "localhost:27003."

A short while later, Figure 5-5 shows that the state of our new member is improving. Finally

Figure 5-6 shows that the member is consistent with the "MASTER" and has now taken the role of a

"SECONDARY" member. With a start time of 18:13:15, Figure 5-6 shows that MongoDB achieved

consistency on the new node at 18:13:48, or 33 seconds.

```
"_id" : 3,

"name" : "localhost:27003",

"health" : 1,

"state" : 3,

"stateStr" : "RECOVERING",

"uptime" : 29,

"optime" : Timestamp(1355679497000, 1),

"optimeDate" : ISODate("2012-12-16T17:38:17Z"),

"lastHeartbeat" : ISODate("2013-01-06T18:13:44Z"),

"pingMs" : 0,

"errmsg" : "syncing to: localhost:27001"
```

Figure 5-5. Excerpt from the "rs.status" command, showing statistics for the new replica set member "localhost:27003" as it begins to persist the data from the "MASTER" member.

```
"_id" : 3,

"name" : "localhost:27003",

"health" : 1,

"state" : 2,

"stateStr" : "SECONDARY",

"uptime" : 33,

"optime" : Timestamp(1355679497000, 1),

"optimeDate" : ISODate("2012-12-16T17:38:17Z"),

"lastHeartbeat" : ISODate("2013-01-06T18:13:48Z"),

"pingMs" : 0
```

Figure 5-6. Excerpt from the "rs.status" command, showing statistics for the new replica set member "localhost:27003" as it achieves a consistent state. Note the "stateStr" property value has updated to "SECONDARY."

Isolation Results

To test the isolation of transactions I opened two shell sessions connected to the same MongoDB instance. I then executed the first two statements shown in Figure 5-7 (one on each shell) simultaneously. These update statements targeted the "postings" collection (used previously) which contained 327,951 documents, and were designed to set the "isolated" property to a value of either "A" or "B." As Figure 5-7 shows, 158,096 documents were updated to have an "isolated" property value of "A," with the remaining 169,855 documents having an "isolated" property equal to "B."

```
> db.postings.update({},{$set:{isolated:"A"}},{multi:true})
> db.postings.update({},{$set:{isolated:"B"}},{multi:true})
> db.postings.find({isolated:"A"}).count()
158096
> db.postings.find({isolated:"B"}).count()
169855
```

Figure 5-7. Competing update commands working against the "postings" collection.

Durability Results

MongoDB stores all write operations to data in RAM. This data is persisted to disk every 100 milliseconds (by default). Keeping the data in-memory provides a boost to performance. However, this does expose the in-memory data to a potential "plug-out-of-the-wall" event.

To test this aspect of ACID, I ran a test where a document "_id: killTest" was inserted, and then immediately killed all mongod server processes. After a restart, the test document "_id: killTest" was still found. In this case, MongoDB managed to persist the in-memory data, even though the server processes were abruptly killed.

Atomicity Interpretation

MongoDB guarantees atomicity for write transactions run against a single document. However,

when it comes to a statement that may update multiple documents in a collection, atomicity is not ensured. When attempting to write to multiple documents at once, the database will not "roll back" to the original state if the transaction aborts.

As shown in Figure 5-1, I ran a code segment which from the MongoDB shell which created my initial dataset. In an atomic, ACID-guaranteed, two-phase-commit database environment, the number of records in the collection should still be zero. This is because the looping operation was terminated and did not successfully complete. However, in this case (as shown in Figure 5-2), the postings collection contained more than three hundred-thousand documents.

I then attempted to apply an update to the documents in the "postings" collection. At the time the operation was killed, 94,208 documents had been updated. By virtue of the fact that a query could identify the affected documents, an atomic rollback (to the original state of the data) did not occur. This experiment clearly demonstrates that MongoDB does not handle multi-document atomicity.

Consistency Interpretation

In my consistency experiment, I had created two MongoDB replicas with 180mb of test data. The test data was created in a manner similar to how it was created for the isolation experiment. I then created a third replica and brought it into the set. The data in Figures 5-4, 5-5, and 5-6 show that it took about 30 seconds for the replica to achieve consistency.

Assume that we had a document in MongoDB with "_id" of "1" and a "testProperty" property of "A." Let's also assume that the document "_id: 1" on our new replica set member was stale, and had a "testProperty" property of "B." During the time elapsed while the new replica was recovering to a consistent state, any queries for "_id: 1" would have yielded the correct "testProperty" value of "A."

Conversely, some NoSQL databases (like Cassandra) are AP (highly-available and partition-

tolerant) databases, which allow for "eventual consistency." MongoDB does provide configuration

options that allow the consistency constraints to be modified. This allows the DBA to systematically

trade consistency for availability as-required by the application. One such way this is done, is for a

DBA to configure a replica set to allow read operations on its "SECONDARY" members. This is done

(10gen, 2012) by executing a "rs.slaveOk()" command from the MongoDB shell.

In the example above, had I configured the replica set with "rs.slaveOk()," it still would have

accepted read requests while working to become consistent. This includes replica set members that are

still "RECOVERING." Depending on how soon after the new member was started, queries for the

document "_id: 1" may have yielded a "testProperty" value of "B" instead of "A." So while MongoDB

technically carries a "CP" CAP configuration, like much else in MongoDB this designation can be

negotiated by an experienced DBA.

Isolation Interpretation

As shown in Figure 5-7, document updates succeeded for both statements that were run. In an

ACID-guaranteed transaction environment, the "isolated" property on each document in the collection

should all be equal to either "A" or "B," but not a combination of both. This simple experiment shows

that MongoDB transactions do not support isolation.

However, transactions can be configured to be isolated (10gen, 2012) at runtime through the use

of the "$atomic" isolation operator. This operator allows updates (that affect multiple documents) to

lock their data, preventing it from being changed by competing transactions. If this operator is not

used, other transactions may modify the targeted data before, after or during the primary update

transaction.

To test the use of the "$atomic" operator, the same statements were run again, but with the

"$atomic:1" statement in the query section. As Figure 5-8 shows, the statements were isolated. The

first statement (setting the "isolated" property to "A") ran first, followed by the second statement. This

resulted in all of the documents having an "isolated" value of "B."

```
> db.postings.update({$atomic:1},
{$set:{isolated:"A"}},{multi:true})
> db.postings.update({$atomic:1},
{$set:{isolated:"B"}},{multi:true})
> db.postings.find({isolated:"A"}).count()
0
> db.postings.find({isolated:"B"}).count()
327951
```

Figure 5-8. Competing update commands updating the "postings" collection, using "$atomic."

It should be noted (10gen, 2012) that using the "$atomic" operator does not provide "all-or-

nothing" atomicity, it only ensures transaction isolation. Data written by transactions that end in a

partially-completed status will persist. This is important to understand, because MongoDB only

supports single-document atomicity, but enforces runtime isolation through the use of the "$atomic"

operator.

Durability Interpretation

In my durability experiment, MongoDB managed to persist the in-memory data, even though

the server processes were abruptly killed. How was this possible? As the MongoDB Manual explains

(10gen, 2012, ch. Administration – Journaling, para. 1), default journaling was hard at work in this

case:

> MongoDB uses *write ahead logging* to an on-disk journal to guarantee write
>
> operation durability and to provide crash resiliency. Before applying a change to the data files,
>
> MongoDB writes the change operation to the journal. Then, if MongoDB should terminate or
>
> encounter an error before it can write the changes from the journal to the data files, MongoDB

can re-apply the write operation and maintain a consistent state.

The journaling option is enabled by default in (64-bit) MongoDB. This option can also be disabled (Hills, 2011) to increase write performance. 10gen warns against this, indicating that (10gen, 2012) a database repair will need to be run from an unaffected replica set member, should a database (running without journaling) experience a power failure.

Keeping this in mind, the experiment above was repeated, this time running all mongod processes with the "--nojournal" option enabled. A test document of "_id: killTest2" was inserted, and then all mongod processes were immediately killed. On an attempt to restart any member of the replica set, the following message was displayed: "[initandlisten] exception in initAndListen: 12596 old lock file, terminating."

Thinking that this MongoDB instance had finally failed a durability test, I went and deleted the lock files in-question, restarted the mongod processes, and queried for the "_id: killTest2" document. It was indeed returned. So while MongoDB may have been known for (Hills, 2011) issues with durability, I could not fault it with a few simple tests.

Summary

The goal of this chapter was to provide an in-depth look at some of the concerns over ACID transaction support, voiced by those reticent to utilize MongoDB as a database server. Depending on the requirements of the data, the level of ACID (that MongoDB is capable of) may or may not be sufficient. While MongoDB does offer certain levels of atomicity and isolation; it does not appear to be on-par with a fully ACID compliant RDBMS. It should be noted that NoSQL databases differ widely in architecture, and this directly affects the level of ACIDity afforded to their operations.

Chapter 6 - Other Considerations with MongoDB and Oracle

Performance and adherence to ACID are central concerns in choosing an E-commerce database. However, there are additional aspects that may influence the choice of database implementation. Some of the important aspects to consider are data security, delivered toolset, and backup/recovery tools.

Security

While not an ACID property, security is an aspect of database processing that is on the forefront of everyone's mind. Given how sensitive data may need to be stored in an e-commerce database, a discussion on security is warranted. Depending on the sensitivity of the data to be stored and the location (both physical and network) of the database server, available security options may make or break a decision to implement a specific database.

Security is something that Oracle 11g R2 does very well. In fact, assigning passwords to the initial system users is part of the installation process. It has a complex system where users are granted permissions to their Oracle schema and other specific database objects. Oracle 11g also has roles that can be created and assigned permissions, then a role can be granted to a user to convey those permissions.

However, MongoDB does not provide the same granularity of security, and does not enable any type of security or authentication by default. MongoDB has security options that can be enabled, but users are recommended to (Banker, 2012, p.226) "take advantage of security features of modern operating systems to ensure the safety of their data." Essentially, this means MongoDB servers should use a local software firewall (typically provided by the operating system) to guard unused ports, and limit both physical and remote access only to verified users of the host system.

One important point to note is that data between MongoDB machines (replica set members, shards, etc…) is sent (Banker, 2012, p. 226) "in the clear." As of MongoDB 2.2, SSL encryption can

be enabled (Merriman, 2012) with a slight degradation in performance. Therefore, running MongoDB in a secure (Chodorow, Dirolf, 2010) trusted environment behind a (hardware) firewall, is highly recommended.

To enable authentication in MongoDB, an administrative user must be created and (Banker, 2012) added to the "admin" database. Then the mongod processes can be started with the "--auth" option to require all connections to have an authenticated user. Authenticated users can then be added to each database, and may be designated as "read only" if required.

Backup and Recovery

For backup and recovery tools, there are few products capable of competing with Oracle Recovery Manager (RMAN). Introduced (Kuhn, 2010) with Oracle Database 8i, RMAN offers a robust, feature-rich toolset that is included by default with both Oracle 11g R2 Standard Edition and Enterprise Edition. Some of the main features of RMAN include but are not limited to (Kuhn, 2010, p. 457):

- Management tools for the tracking, deletion and obsoleting of backup files.

- Incremental backups.

- Block-level recovery.

- Compression and Encryption.

- Validation and testing of backup files.

- Data conversion between multiple platforms.

- Recovery solution assistance from the Data Recovery Advisor.

One method of performing a MongoDB backup (Chodorow, Dirolf, 2010, p. 121) is to create "a copy of all files in the data directory." However, copying the data directory of a running instance of MongoDB will likely produce a corrupted backup containing errors. Of course, (Chodorow, Dirolf, 2010) the server can be shut-down before taking the backup copy, but this is certainly not preferred.

MongoDB also has the "mongodump" and "mongorestore" tools. For comparison, it helps to think of these tools as the rough equivalents of Oracle's Data Pump. While not as feature-rich as Oracle's RMAN, mongodump and mongorestore can be used (Membry, et-al, 2010) to backup and restore entire databases or even individual collections. The MongoDB backups can also be automated (Membry, et-al, 2010) through the use of shell and JSON scripts.

As an additional way to fulfill this need, 10gen announced in April of 2013 the addition of the MongoDB Backup Service. The MongoDB Backup Service is "a cloud-based solution geared toward its customers who use large data sets" (Backaitis, 2013, para. 1). This solution is billed as being fast, convenient and cost-effective, while providing (among other features) security, high-availability and point-in-time recovery. While this service was announced too close to the conclusion of this project to be able to ascertain its effectiveness, it does show that 10gen is serious about providing its users with better options.

Delivered Toolset

As most NoSQL database packages tend to concentrate on the database product itself, there is typically little in the way of delivered toolsets. However, MongoDB does provide tools that allow users to perform complex Map/Reduce aggregations, in addition to geospatial indexes for location-based data. Other tools such as text searching and regular expression matching (which are not features in many other NoSQL databases) are also included with MongoDB.

On the other hand, Oracle 11g R2 comes with plenty of delivered tools. Features of Oracle 11g

R2 include (but are not limited to) tools like (Nanda, 2008) Advanced Hybrid Columnar Compression, Automatic Storage Management (ASM), SQL Performance Analyzer, RMAN, and the Oracle Exadata Simulator. Additionally, Oracle 11g R2 delivers an extensive set of tools supporting business intelligence (BI) and analytics. Business analysts will find features like (Nanda, 2008) Cube-Organized Materialized Views, Partition Change Tracking (for materialized views), and the Analytic Workspace Manager to be quite useful. The Query Rewrite function also helps business users, as it checks for queries data that match data in existing materialized views, and utilizes that data without performing an operation on the actual table.

Summary

The goal of this chapter was to examine additional elements to be considered in the selection of a database. MongoDB does have security options available, even though they are not enabled by default. Data communications can be encrypted by SSL, and users can be required to authenticate on each database. However, a definite strength of Oracle 11g R2 is the granularity and level of security provided.

In dealing with a "big data" solution, whichever database you choose will require daily backups and periodic restores. The backup and recovery tools will need to be able to serve both the everyday and emergency needs of the DBA. As with security, the extra features provided with Oracle 11g R2 are certainly valuable options to have.

The delivered toolset that accompanies a database product is certainly not the prime reason for selecting it. However, having a good, pre-packaged toolset can certainly be an asset.

Oracle 11g R2 has a very robust set of utilities which offer functionality to DBAs, developers, and business analysts. While MongoDB does not offer a similar quantity of features, it does provide more additional tools than many of its NoSQL competitors.

Chapter 7 - Conclusions and Discussion

Conclusions

In the first chapter of this paper, three research questions were presented. These questions helped to drive the direction of the experiments, and provide focus for the goals of this paper. The three questions were posed:

1. Between MongoDB and Oracle, which will perform faster e-commerce based transactions?

2. Between MongoDB and Oracle, which will provide greater data integrity (based on ACID standards)?

3. Is there an acceptable level of compromise between performance and data integrity to allow for an optimal e-commerce solution on MongoDB?

Additionally, answers to those questions were also hypothesized. This hypothetical answers help to provide perspective to the results, so that their relative position of reality to perception can be ascertained. The hypothesized answers were given:

1. MongoDB will out-perform Oracle in terms of raw read and write speeds.

2. MongoDB will fail to adequately perform multiple ACID tests.

3. An appropriate trade-off of ACID for performance will be able to be made to justify the use MongoDB in an e-commerce environment.

As the results of the performance tests have shown, MongoDB did out-perform Oracle in almost every test. The two instances of Oracle out-performing MongoDB (the top and second level product navigation queries) were negated after altering the MongoDB schema to a more-optimal configuration. The difference in customer password update performance times proved negligible (0.2841ms for

MongoDB and 0.8465ms for Oracle). However, the remainder of the read and write performance heavily-favored MongoDB, making the first hypothesized answer correct.

MongoDB's performance in a series of ACID-based tests was discussed in the "Results and Evaluation of the MongoDB ACID Experiments" chapter. Using the default settings, MongoDB exhibited adequate performance in both consistency and durability. While it may have demonstrated some weakness in the isolation-based tests, the update statements were demonstrated as capable of being configured to be isolated at runtime.

MongoDB did not fare well in the atomicity-based tests, and this may preclude its use in scenarios where transaction guarantees (with rollback) are required. As author Kyle Banker points out (Banker, 2012, p. 256) "no equivalent of RDBMS's BEGIN, COMMIT, and ROLLBACK semantics exists" in MongoDB. If the application requirements exhibit characteristics for which MongoDB is a good fit, but needs to provide some level of transactional guarantees, then those requirements will need to be addressed at the application level.

While MongoDB failed some of the ACID tests in this study, it did perform better than expected. As previously mentioned, the initially-failed tests for isolation passed given additional (runtime) configuration. But overall, the second hypothesized answer remains correct.

Considering what has been summarized here in-reference to MongoDB's performance relative to ACID, the validity of the third hypothesized answer remains difficult to assess. What is clear is that whether or not MongoDB's default level of ACID adherence is acceptable will depend on the requirements of the overall application. If the data or application (and more importantly, its users) can allow some level of read/write failure, then providing multi-document atomicity may not be a concern. But if atomic transactions are required and are not possible to implement at the application level, then MongoDB is not be a valid choice for that particular application. Therefore the best answer for the

third hypothesized question is "it depends."

Summary of Contributions

The performance experiments executed clearly demonstrate that MongoDB is very fast. In a simulated environment, MongoDB out-performed Oracle 11g R2 in its ability to handle many e-commerce transactions over a short period of time. Oracle performed well, albeit not as well as MongoDB. In most cases, performance on Oracle still proved to be adequate for e-commerce transactions.

My experiments designed to examine MongoDB's adherence to ACID transactions produced concrete results. In relation to my hypothesized question (and answer) concerning ACID transactions, MongoDB performed as expected in the ACID experiments. It passed tests for both consistency and durability. The atomicity test proved to be a failure, and the isolation test showed that it could succeed with additional configuration. These experiments provided specific data about how well MongoDB adheres to each property of ACID.

These findings indicate that the performance gains achieved by MongoDB do come at the cost of (some) ACID properties and transaction guarantees. A good question to ask would be "does my e-commerce website really require the performance of MongoDB?" In the case of the order insert experiment, MongoDB's performance gain was about 20ms. As Google's recommended page load time is (Eisenberg, 2011) two seconds (2000ms), taking an additional 20ms to insert a record into the database represents only one percent of the total load time. The value placed on that one percent is something that each individual must decide in relation to their own application or system.

Lessons Learned

This section will discuss specific knowledge that was generated from this study. Specific points to be learned from this study include the importance of schema design, security, and delivered toolsets. Likewise, there were additional lessons learned from conducting the regular expression search experiments.

Importance of Schema Design.

The schema for the product and hierarchy data in Oracle was normalized; with the "Product" table containing the product data, the "Hierarchy table containing category data for each hierarchy level, and the "ProductToHier" table containing the relationship data between products and their lower-level hierarchy. The initial MongoDB schema was de-normalized; with the "products" collection containing the product data, the category hierarchy data, and the product-to-hierarchy relationship data.

In the product navigation experiment, the tests run on Oracle initially out-performed the initial tests run for MongoDB in the selection of top and second level product categories. After an alteration in the MongoDB schema, the reruns of the MongoDB tests proved to be much faster. This alteration involved applying a slight amount of normalization to the schema. The data pertaining to the hierarchies and product-to-hierarchy relationships was split-off into a separate "categories" collection. This change resulted in an average transaction time of 0.6701ms, down from 153.061ms for top-level category navigation. The second-level category navigation transaction averaged 0.0085ms, down from 221.619ms.

This finding leads to the conclusion that normalization in MongoDB schema design is more of a "sliding scale," rather than a set of hard and fast rules. While one of the advantages of MongoDB is its (Membry, et-al, 2010) flexible schema, it should be noted that the practice of (Chodorow, 2011a, ch. 1) "Duplicate data for speed" should not be interpreted as always needing to apply the maximum amount of de-normalization. In the case of these experiments, MongoDB performed significantly faster when an appropriate amount of normalization was applied.

Security.

One advantage of RDBMS's is that they tend to share a similar security model. All connections

to the database must be authenticated and typically each application will have its own user id. Users

are created and assigned permissions on a specific table in a database with specific privileges.

Database system maintenance operations are run as a privileged user id, usually a system or DBA

account.

When compared to a RDBMS like Oracle 11g, security in MongoDB (and in many NoSQL

databases) leaves a lot to be desired. While MongoDB does not enable any security by default, user

authentication and SSL encryption options can be activated. However, those features do not provide

the same granular level of security as Oracle does.

Author Daniel Doubrovkine points out that Chodorow and Dirolf's recommendation of running

MongoDB in a "trusted" network environment may not be enough (Doubrovkine, 2011, para. 10):

> Common sense should tell us that we cannot be storing sensitive data in any type of storage that
>
> is protected by the network alone. It's not acceptable for traditional RDBMS, so it shouldn't be
>
> acceptable for any other remote storage, including NoSQL.

Doubrovkine describes MongoDB's user authentication option as a "good start." But he

proceeds on to illustrate issues with the segregation of permissions." He recommends (Doubrovkine,

2011, para. 22) that "before you switch, evaluate your risks."

It should be noted that MongoDB has implemented support for Kerbreros Authentication

(10gen, 2012) as of development release 2.3. Adhering to Kerbreros authentication standards certainly

helps increase security. Each organization considering using MongoDB should give it a careful

evaluation to ensure that its security requirements are met.

Delivered Toolset.

Examination of extra features is typically a secondary consideration in choosing a database. But choosing one with useful delivered tools can save an organization from having to develop or purchase them. While offering more robust toolsets than MongoDB in many areas, it is important to remember that Oracle has the benefit of 35 years of product development behind it. By comparison, MongoDB is less than five years old. Given MongoDB's growing popularity and (Xavier, 2013) 10gen's recent partnership with IBM, it is reasonable to assume that helpful and robust toolsets will eventually be developed. Currently Oracle 11g R2 delivers a vastly superior toolset, supporting a variety of administrative and analytical functions.

Regular Expression Searching.

The regular expression search tests proved to illustrate the biggest performance gap between MongoDB and Oracle 11g R2. This indicates some level of bias in this test, in that regular expression searching is not a strong-suit of Oracle 11g R2. Additionally, my conclusions and recommendations from this test do not favor Oracle or MongoDB, rendering it moot.

Recommendations and Future Research

This section will briefly describe recommendations based on one of the experiments in this study. It will also identify a few areas for possible future research. These points are based on questions raised by the results of this study that were deemed to be out of scope for this project.

Recommendations.

One of the experiments performed in this study was a keyword search which utilized each database's regular expression library. As shown in the chapter titled "Performance Results and Evaluation," MongoDB averaged a transaction time of 4937 nanoseconds, compared to 194.4144 milliseconds for Oracle. While tempted to count this as a success for MongoDB, in the case of an e-

commerce website implementation, I would recommend neither MongoDB nor Oracle for that functionality.

Modern e-commerce websites require a rich toolset capable of delivering keyword search capabilities, as well as guided navigation and keyword spotlighting. Additional features like auto-complete, runtime application of custom thesaurus entries and spelling corrections (e.g. "did you mean") are also highly desirable. Robust, enterprise search products exist with all of these features, including proprietary solutions like Oracle Endeca Commerce, as well as open-source products like Apache Solr.

Enterprise search products are written by engineers who are highly-specialized in the design of search algorithms. It is always a better option to go with a delivered search solution, rather than trying to build your own via regular expressions or full text search implementations. In response to MongoDB releasing a recent (development) version with improved text searching capability, codecentric AG author Tobias Trelle arrived at a similar conclusion (Trelle, 2013, para. 15): "Of course, this implementation of a full text search won't enable MongoDB to compete with search engines like Apache Solr or Elastic Search, but it is a step in the right direction."

Possible Future Research.

It was observed in a few of the experiments that additional factors may have influenced the results. Further study on those effects may bring to light new observations to increase the overall understanding why each database performs as well as it does under certain conditions. One such additional topic for a more-focused study would be a more in-depth comparison of the regular expression performance and capabilities of both Oracle 11g R2 and MongoDB. It was noted that these databases utilize different implementations of regular expression libraries, with Oracle using the POSIX library and MongoDB using the PCRE library. A more accurate comparison of their performance would have been possible if they both utilized the same library, or if it was possible to

reconfigure them to do so.

On the surface, it is tempting to infer that since MongoDB out-performed Oracle in the regular expression experiments, that the PCRE library likewise out-performs the POSIX library. However, the results show that MongoDB ultimately exhibited faster transaction times in every performance experiment, which would indicate that the particular library implemented probably made little difference. This raises the question, "if the regular expression library implementations were the same, how much (if any) performance difference would it make?"

For the performance tests, MongoDB and Oracle 11g R2 were configured to run as close to the default settings (for each database) as possible. This means that MongoDB was running without any security measures in place, and poses several questions for further research: "How fast would MongoDB perform with user authentication and/or SSL enabled?" Dwight Merriman (Chairman and co-founder of 10gen) has stated (Merriman, 2012) that enabling SSL on communication between MongoDB machines would result in "slight" performance degradation. This raises the question of "how much (exactly) performance would MongoDB have to sacrifice for SSL?"

In addition, it has been noted where experiments (for both MongoDB and Oracle) would need to be altered in a true e-commerce environment to achieve compliance with PCI Standards. Additional processing would need to take place to ensure that all passwords and payment data was properly encrypted upon its entrance into the application. "By how much would that additional processing affect performance?"

Summary

The goal of this study was to apply MongoDB to an e-commerce environment, and measure its performance against a RDBMS. Being a well-known product utilized by many enterprise customers around the world, Oracle was chosen as the RDBMS platform. MongoDB was shown to perform well,

but did not successfully negotiate all of the ACID transaction support experiments. Despite this, I

believe I have demonstrated that MongoDB certainly has appropriate use cases.

MongoDB has shown that it is very fast. Its flexible schema is certainly an attractive option for

storing data rows (documents) which may not all have the same properties. If your application needs to

be able to scale to support large amounts of data, and transactions do not require full ACID

compliance, MongoDB is indeed a viable solution.

As for an e-commerce framework, MongoDB is well-suited for storing diverse product data or

even customer records. But a RDBMS like Oracle 11g R2 offers many intrinsic qualities that make it

desirable for storing sensitive data like ordering and payment information. In an ideal world where a

project architect has both the freedom and the resources to choose the right tool(s) for the job, building

these systems on separate database architectures may make the most sense.

In Appendix H of this paper, I have written a questionnaire, designed with the purpose of

helping readers ascertain which type of database architecture may best suit their needs. It is essentially

a multiple-choice survey, with the points of each letter being checked against a certain threshold. Each

letter corresponds to a type of database architecture. If a certain threshold for an architecture is met,

then the associated architecture should be considered.

It is possible for the scores from the questionnaire to indicate that the user should investigate

multiple database architectures. That underscores the point that there is not a black-and-white, absolute

solution to the problems posed by Brewer's CAP Theorem. Author and consultant Julian Browne

(Browne, 2009, para. 25 & 26) provides some insight on this point by stating that:

If you have to drop one of consistency, availability, or partition tolerance, many opt to drop

consistency which is the raison d'être of the database. The logic, no doubt, is that availability

and partition-tolerance keep your money-making application alive, whereas inconsistency just feels like one of those things you can work around with clever design.

Like so much else in IT, it's not as black and white as this. Eric Brewer, on slide 13 of his PODC talk, when comparing ACID and its informal counterpart BASE even says "I think it's a spectrum."

Browne makes another good point shortly thereafter (Browne, 2009, para. 26) when he writes "Nobody should interpret CAP as implying the database is dead." As shown by Amazon's Dynamo project, there are many problems of scale where NoSQL solutions (DeCandia, et-al, 2007) have provided desired levels of availability, fault-tolerance and performance. But many applications still require the storage of relational data, sensitive data requiring security, and/or the ability to perform business analytics and ad-hoc queries. These are problems which relational databases continue to solve well.

It is my conclusion that the decision of which database architecture to use is one that should be made in the early stages of application/platform design. That decision will be influenced by answering questions about the data, the expected behavior of the application, and ultimately the business requirements. It may be in the project architect's best interests to apply the CAP theorem to the application itself, to gain a better understanding of the optimal database architecture to select. While MongoDB and other NoSQL databases are great tools that can solve interesting problems, you should have a specific reason for choosing one over a (RDBMS) solution which has the benefit of 40 years of computer science research behind it.

Chapter 8 - Works Cited

10gen. (2012). *MongoDB Manual.* 10gen, Inc. New York, NY. Retrieved from:

    http://docs.mongodb.org/manual/about/.

Backaitis V. (2013). *10Gen Introduces MongoDB Backup Service.* CMS Wire. Simpler Media Group,

    Inc. Retrieved from: http://www.cmswire.com/cms/information-management/10gen-

    introduces-mongodb-backup-service-020737.php.

Banker K. (2010). *MongoDB and E-commerce.* Retrieved from:

    http://kylebanker.com/blog/2010/04/30/mongodb-and-ecommerce/.

Banker K. (2012). *MongoDB In Action.* Manning Publications. Shelter Island, NY.

Brewer E., Fox, A. (1999). *Harvest, Yield, and Scalable Tolerant Systems.* University of California at

    Berkeley. Berkeley, CA. Retrieved from:

    http://lab.mscs.mu.edu/Dist2012/lectures/HarvestYield.pdf.

Browne J. (2009). *Brewer's CAP Theorem.* Retrieved from:

    http://www.julianbrowne.com/article/viewer/brewers-cap-theorem.

Casteel J. (2007). *Oracle 10g SQL. Thomson Course Technology.* Boston, MA. (pp. 142, 328-331).

Chodorow K., Dirolf M. (2010). *MongoDB: The Definitive Guide.* O'Reilly Media, Inc. Sebastapol,

    CA.

Chodorow K. (2011a). *50 Tips & Tricks for MongoDB Developers.* O'Reilly Media, Inc. Sebastapol,

    CA. Kindle Cloud Reader edition. Retrieved from: Amazon.com.

Chodorow K. (2011b). *Scaling MongoDB.* O'Reilly Media, Inc. Sebastapol, CA. Kindle Cloud

    Reader edition. Retrieved from: Amazon.com.

Coronel C., Rob P. (2007). *Database Systems: Design, Implementation, and Management.* (ch. 5)

Course Technology (8th edition). Boston, MA.

Decandia G., Hastorun D., Jampani M., Kakulapati G., Lakshman A., Pilchin A., et-al. (2007).

Dynamo: Amazon's Highly Available Key-value Store. Proceedings of the Twenty-First

ACM SIGOPS Symposium on Operating Systems Principles (pp. 205-220). doi:

10.1145/1294261.1294281

Doubrovkine D. (2011). *Is 2011 the Year of NoSQL Data Breaches?* Wired Business Media.

Retrieved from http://www.infosecisland.com/blogview/11411-Is-2011-the-Year-of-NoSQL-

Data-Breaches.html.

Eisenberg B. (2011). *How Your Website Loses 7% of Potential Conversions.* ClickZ: Marketing News

& Expert Advice. Retrieved from: http://www.clickz.com/clickz/column/2097323/website-

loses-potential-conversions.

Freire P. (2007, May 4). *PHP: PCRE Functions - PCRE faster than POSIX RE? Not always.* [Msg 6]

Posted to: http://www.php.net/manual/en/ref.pcre.php#74939.

Gilbert S., Lynch N. (2002). *Brewer's Conjecture and the Feasibility of Consistent, Available,*

Partition-Tolerant Web Services. Laboratory for Computer Science - Massachusetts Institute of

Technology. Cambridge, MA. doi: 10.1145/564585.564601

Grobler H. (2011). *Using MongoDB for an E-Commerce Platform.* Strategic Worldwide Applications

and Technologies. Retrieved from:

http://www.scribd.com/fullscreen/60323134?access_key=key-pulkfulpfr9ozyupebb.

Hess K. (2010). *Top 10 Enterprise Database Systems to Consider.* ServerWatch. IT Business Edge.

Retrieved from: http://www.serverwatch.com/article.php/3883441.

Hills A. (2011). *MongoDB Journaling Performance – Single Server Durability.* Retrieved from:

http://www.adathedev.co.uk/2011/03/mongodb-journaling-performance-single.html.

Kovacs K. (2012). *Cassandra vs. MongoDB vs. CouchDB vs. Redis vs. Riak vs. HBase vs.*

Membase vs. Neo4j comparison. Retrieved from:

http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis/.

Kuhn D. (2010). *Pro Oracle Database 11g Administration.* Apress. New York, NY.

Lerdof R., Tatroe K. (2002). *Programming PHP.* O'Reilly Media, Inc. Sebastapol, CA.

Maddock J. (2003). *Regular Expression Performance Comparison.* Retrieved from:

http://www.boost.org/doc/libs/1_51_0/libs/regex/doc/gcc-performance.html.

Membry P., Plugge E., Hawkins T. (2010). *The Definitive Guide to MongoDB: The NoSQL Database*

for Cloud and Desktop Computing. Springer-Verlag. New York, New York, NY. Kindle

Cloud Reader edition. Retrieved from: Amazon.com.

Moradi M. (2011). *Essential E-Commerce Website Features: Tips and Examples.* Retrieved from:

http://sixrevisions.com/user-interface/e-commerce-website-features-tips-examples/.

Mullins, C. (2011). *The Database Report – July 2011.* The Database Administration Newsletter.

Retrieved from: http://www.tdan.com/view-articles/15299.

Nanda A. (2008). *Oracle Database 11g: The Top Features for DBAs and Developers – Data*

Warehousing and OLAP. Oracle Corporation. Retrieved from:

http://www.oracle.com/technetwork/articles/sql/11g-dw-olap-100058.html.

Oliver A. (2012). *Ill-informed Haters Go After MongoDB.* InfoWorld. Retrieved from:

http://www.infoworld.com/d/application-development/ill-informed-haters-go-after-mongodb-

205096?page=0,1.

PCI Security Standards Council (2010). *PCI DSS Quick Reference Guide.* Retrieved from:

https://www.pcisecuritystandards.org/documents/PCI%20SSC%20Quick%20Reference%20Gui

de.pdf.

Song I., Whang K. (2000). *Database Design for Real-World E-Commerce Systems.* IEEE Data

Engineering Bulletin, March 2000, Vol 23.

Tindel C. (2012). *Intro to Schema Design. MongoDB – Chicago (lecture/presentation).* 10gen.

Chicago Hyatt Magnificent Mile, Chicago, IL. Retrieved from:

http://www.10gen.com/presentations/mongodb-chicago-2012/intro-schema-design.

Trelle T. (2013). *MongoDB Text Search Explained.* codecentric AG. Retrieved from:

http://blog.codecentric.de/en/2013/01/text-search-mongodb-stemming/.

Yu H., Vahdat A. (2000). *Design and Evaluation of a Continuous Consistency Model for Replicated

Services.* Department of Computer Science – Duke University. Durham, NC. OSDI'00

Proceedings of the 4th Conference on Symposium on Operating System Design &

Implementation.

Xavier. J. (2013). *IBM and 10Gen team up to dominate the database market.* Silicon Valley Business

Journal. Retrieved from: http://www.bizjournals.com/sanjose/news/2013/06/04/mongodb-and-

ibm-team-up.html.

Appendix A: Definition of Terms

Atomicity – A guarantee that a transaction run against a database will either complete entirely or not at all. This prevents an update from producing a state of "partial completeness."

Consistency – Ensuring that the same data is written to all nodes in a partitioned database system. Also, data which is consistent is that which does not violate any integrity constraints of the database.

Database Transaction – An operation (read or write) executed against the database.

De-normalization – The process of designing a database schema to selectively implement redundant data to increase performance.

Distributed Systems – A general term for an application system that spans multiple computers. It is commonly used to represent many types of application systems, including (but not limited to) web services and databases.

Durability – The guarantee that data written to the database will persist in the event of a "plug out of the wall" scenario.

Eventual Consistency – A data processing strategy that sacrifices strong-consistency across nodes in exchange for high or consistent availability.

Horizontal Scaling – A technique designed to increase performance in a distributed system by adding additional replicas of the data.

Isolation – The guarantee that a transaction run against a database does not affect the outcome(s) of other transactions that currently running.

Map/Reduce – A process of data aggregation, whereby data is stored in a key-value map data structure, and the data set is "reduced" into a smaller list based-on certain user-defined criteria.

mongod – The name of the MongoDB database server process.

Normalization – The process of structuring database schema design with the goal of eliminating redundant data, transitive and partial dependencies, with the goal of (Coronel, Rob 2007, ch. 5) "reducing the likelihood of data anomalies."

Oracle Schema - (Casteel, 2007) The database objects owned by a specific user. These objects not limited to tables, but to all other objects related to the tables. Use of "schema" in the context of Oracle is specifically addressed in Chapter 2.

Quorum – For NoSQL databases such as Cassandra, quorum is a configurable parameter which helps determine the number of nodes (usually slightly more than half) required for data consistency before responding to the client.

Replica – A MongoDB server which is a copy of the "primary" database server and runs in a "secondary" configuration.

Schema – A database's tables and relationships between those tables. This typically includes field or property names, as well as data types.

Shard – A node that stores and serves a distinct subset of partitioned data in a MongoDB shard cluster.

Shard Cluster – A group of MongoDB servers configured as shards, and not replicas.

Shard Key – A property chosen to evenly split partitioned data in a MongoDB shard cluster.

Sharding – The act of separating and distributing data in a MongoDB shard cluster. The individual shards do not share replicated data. Data stored on each shard is unique, and is determined by the shard key.

Transaction – An operation (read or write) in a distributed system.

Two-Phase-Commit – A type of transaction involving multiple transactions executing against various parts of the database. If one of the multiple transactions should report a failure, then each subsequent transaction will be cancelled and each prior transaction will be rolled-back. This helps to ensure atomicity and prevent data peculiarities.

Vertical Scaling – A technique designed to increase performance in a distributed system by increasing the available hardware resources (example: adding more RAM).

Appendix B: Annotated Bibliography

10gen. (2012). <u>MongoDB Manual.</u> 10gen, Inc. New York, NY. Retrieved from:

<u>http://docs.mongodb.org/manual/about/</u>.

The official MongoDB online instruction manual is versioned in GitHub and is a collaborative effort by 10gen and the MongoDB community. This manual covers several aspects of MongoDB configuration, development and behavior. I utilized this source several times throughout this project, especially during the description and testing of MongoDB transactions and how they relate to ACID transactions.

Anderson J.C., Lehnardt J., Slater, N. (2010). <u>CouchDB: The Definitive Guide.</u> O'Reilly Media Inc., Sebastapol, CA.

While (obviously) tailored to the CouchDB NoSQL database, this book offers excellent descriptions on the problems that NoSQL databases were designed to overcome. The chapter on Eventual Consistency speaks to these technical limitations, including an in-depth description of Local Consistency. Essentially CouchDB satisfies the CAP theorem (Consistency, Availability, Partition tolerance) by sacrificing consistency, and directs its available nodes to varying versions of records (or documents).

Banker K. (2010). <u>MongoDB and E-commerce.</u> Retrieved from:

<u>http://kylebanker.com/blog/2010/04/30/mongodb-and-ecommerce/</u>.

Author Kyle Banker discusses some of the misconceptions about MongoDB and its suitability for e-commerce. He breaks down his article into sections representing common e-commerce scenarios, as well as how to use MongoDB as a solution. Often, Banker illustrates how straight-forward the data becomes, as the need for executing a join no longer exists. After explaining a little about MongoDB's

limitations concerning transactions, the author concludes by mentioning several positive points for

MongoDB (in terms of use in e-commerce) and even calls its use for web content management a clear

win. It should be noted that Banker has publicly retracted some of the statements made in this article,

stating he no longer agrees that MongoDB should be used for web order and payment operations, or

any other system requiring atomic transactions.

Banker K. (2012). <u>MongoDB In Action.</u> Manning Publications. Shelter Island, NY. (pp. 256).

Author Kyle Banker takes a three-phased approach to explaining MongoDB, first exploring the

data internals (JSON/BSON), architecture and installation. Next he delves into developing against

MongoDB, using the Ruby language. In this part, Banker also describes a strategy to design an e-

commerce data model in MongoDB. Finally, the author discusses advanced MongoDB topics,

including replication, sharding and other administration-based tasks. In addition, Banker also provides

useful information on working with MongoDB in other popular languages (PHP, Java, C++).

Brewer E., Fox, A. (1999). <u>Harvest, Yield, and Scalable Tolerant Systems.</u> University of

California at Berkeley.

This paper by Eric Brewer and Armando Fox discusses the challenges which befall a distributed

system, and the ways in which it can attempt to solve them. Most notably, the authors indicate that

distributed systems aim to be consistent, highly-available, and partition tolerant; but that they can

realistically only satisfy two of those conditions. The author refers to these conditions as the Strong

CAP Principle, and they are central to any discussion on NoSQL technology.

Browne J. (2009). <u>Brewer's CAP Theorem.</u> Retrieved from:

http://www.julianbrowne.com/article/viewer/brewers-cap-theorem.

Consultant and author Julian Browne starts by summarizing the important points of the CAP

theorem, as well as why it is significant. Browne frequently uses examples of well-known e-commerce websites that have sacrificed consistency for availability and partition tolerance, and mentions the gains that they have seen as a result. It is important to note that this article does not specifically mention MongoDB, Cassandra or any other NoSQL database. Browne simply seeks to illustrate why the consistent-available approach of RDBMS may not be sufficient in all cases. The author also introduces the informal term BASE (Basically Available Soft-state Eventually consistent), as the logical opposite of ACID (Atomicity Consistency Isolated Durable). Browne indicates that designing a system and deciding between BASE and ACID transactions is not necessarily a black-and-white choice, but rather is representative of points on a spectrum.

Casteel J. (2007). <u>Oracle 10g SQL.</u> Thomson Course Technology. Boston, MA. (pp. 142, 328-331).

Author Joan Casteel takes an iterative approach in discussing SQL for Oracle 10g. The discussions start out on simple SELECT statements and progressing to complex joins, as well as syntax for data definition and data control. The source proved relevant to this project, because of Casteel's description on why transactions are important, as well as the author's discussion on Oracle SQL regular expression functionality and adherence to the POSIX regular expression standards.

Chodorow K., Dirolf M. (2010). <u>MongoDB: The Definitive Guide.</u> O'Reilly Media, Inc. Sebastapol, CA.

Authors Kristina Chodorow and Michael Dirolf take a low-level, nuts-and-bolts approach to describing MongoDB, development strategies and advanced administration. After an introduction, the authors describe how to create, read, update, and delete MongoDB documents (data) via the JSON-based MongoDB command line interface. They then describe indexing, data aggregation, and additional advanced topics like capped collections, GridFS and references. After a discussion on administrative topics like sharding, replication and horizontal scaling, the authors also discuss

application design and provide examples on how they addressed certain problems.

Chodorow K. (2011a). 50 Tips & Tricks for MongoDB Developers. O'Reilly Media, Inc.

Sebastapol, CA.

Author and 10gen software engineer Kristina Chodorow discusses fifty quick tips for developers working with MongoDB. The tips are laid out sequentially, making for an excellent desktop programming reference. Chodorow covers a variety of topics: from application design, implementation and data consistency, just to name a few. While the author's main audience is software developers, the last section covers administration which provides insight on cleaning up chunks collections, running the repair process, sharding and replication.

Chodorow K. (2011b). Scaling MongoDB. O'Reilly Media, Inc. Sebastapol, CA.

Author and 10gen software engineer Kristina Chodorow examines MongoDB's horizontal scaling capabilities. Chodorow covers advanced topics in the areas of configuration and administration of a shard cluster. She also includes a section on troubleshooting entitled "What to Do When Things Go Wrong," which proved useful as I examined how MongoDB handles durability and consistency.

Coronel C., Rob P. (2007). Database Systems: Design, Implementation, and Management. (ch. 5)

Course Technology (8th edition). Boston, MA.

This book describes relational database management systems, in terms of data modeling and querying. In the chapter of relevance to this thesis (chapter 5), the authors discuss data normalization. Specifically, they illustrate the goals behind the elimination of redundant data, partial dependencies, and transitive dependencies. This is significant, because NoSQL advocates take the opposite approach in de-normalizing to achieve performance.

Decandia G., Hastorun D., Jampani M., Kakulapati G., Lakshman A., Pilchin A., et-al. (2007).

Dynamo: Amazon's Highly Available Key-value Store. Proceedings of the Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (pp. 205-220).

This paper, written by several researchers from Amazon.com discusses the creation of the Dynamo NoSQL database. The authors articulate the background, describing some of the unique scaling challenges faced by Amazon. They then present related work that they drew from, as well as their system design and implementation. This paper is significant to this study, because it details how Amazon's researchers and engineers made sacrifices in the areas of data consistency to ensure availability and uptime in an environment so large (10,000-plus servers) that something is always failing.

Doubrovkine D. (2011). Is 2011 the Year of NoSQL Data Breaches? Wired Business Media. Retrieved from http://www.infosecisland.com/blogview/11411-Is-2011-the-Year-of-NoSQL-Data-Breaches.html.

Author Daniel Doubrovkine discusses security (or lack thereof) in NoSQL databases, and brings-up MongoDB in particular. Doubrovkine examines of the common NoSQL recommendation of running only in a "trusted environment" can be exploited. The author calls MongoDB's basic offered authentication a "good start," but insists that it is not enough. Doubrovkine concludes that if a company has developers using an open source NoSQL database, that they should consider contributing stronger security back to it.

Freire P. (2007). PCRE faster than POSIX RE? Not always. Retrieved from: http://www.php.net/manual/en/ref.pcre.php#74939.

Author Pedro Freire, PhD, discusses PCRE and POSIX regular expression libraries in PHP. He challenges the notion that PCRE regular expressions are always faster than their POSIX counterparts, and posts results of a series of experiments examining the two. Freire indicated that although PCRE

expressions out-performed POSIX (sometimes by as much as 100 times), that the results proved to be

inconclusive. Despite PCRE's performance in most instances, there were some scenarios where

POSIX out-performed PCRE, making it difficult to say that one is definitively "better" than the other.

This study proved to be quite helpful to this author in examining the differences in performance

between MongoDB and Oracle regular expression functionality.

Gilbert S., Lynch N. (2002). <u>Brewer's Conjecture and the Feasibility of Consistent, Available,</u> <u>Partition-Tolerant Web Services.</u> Laboratory for Computer Science - Massachusetts Institute of Technology. Cambridge, MA.

In this paper, authors Seth Gilbert and Nancy Lynch prove Brewer's CAP Conjecture for

distributed systems (web services). After defining the parts of CAP, set up logical models for each

under varying conditions: synchronous vs. asynchronous, with or without a timer, etc. They conclude

that it is indeed impossible for a distributed system to be consistent, available and partition tolerant all

at once. However, the authors note that it is possible (for partially synchronous models) to come to a

practical compromise between consistency and availability. This work is important to the core of

NoSQL technology, because it is what essentially proved Brewer's CAP Conjecture to Brewer's CAP

Theorem.

Glover A. (2012). <u>Flexing NoSQL: MongoDB in review.</u> ComputerWorld. Retrieved from: http://www.computerworld.com/s/article/9224061/Flexing_NoSQL_MongoDB_in_review?taxono myId=11&pageNumber=1.

Author Andrew Glover examines the document-oriented NoSQL database MongoDB, and

describes its many features and how it differs from an RDBMS. It touches on MongoDB's query

language; briefly describes searching via Boolean and regular expressions, as well as MapReduce.

Despite the performance and cost advantages to MongoDB, Glover does bring attention to MongoDB's

lack of security, queries that do not support joins, and lack of ACID transactions. This source was

deemed to be important to this study, as it aims a critical point of view at MongoDB and highlights

(what some view as) its shortcomings.

Greenwald R., Stackowiak R., Stern J. (2008), <u>Oracle Essentials: Oracle Database 11g.</u> O'Reilly

Media Inc. (4th edition). Sebastapol, CA.

Authors Rick Greenwald, Robert Stackowiak and Jonathan Stern describe many aspects of the

fundamental architecture of Oracle 11g. They describe how to install, manage and tune Oracle 11g.

The authors also cover some of the newer features of Oracle 11g, including (but not limited to) Real

Application Testing, Advanced Compression, Total Recall, Flashback and transparent data encryption.

Central to the topic(s) of this thesis, however, was the chapter on Oracle Multiuser Concurrency, which

describes locking, transactions, integrity issues as well as levels of isolation.

Grobler H. (2011). <u>Using MongoDB for an E-Commerce Platform.</u> Strategic Worldwide

Applications and Technologies. Retrieved from:

http://www.scribd.com/fullscreen/60323134?access_key=key-pulkfulpfr9ozyupebb.

Author Hennie Grobler examines the specific application of MongoDB for an e-commerce

system. Grobler emphasizes three factors that he states have the most effect in a MongoDB

implementation: schema design, sharding and experience. Grobler's observations on schema design

were found to be helpful by this author in the examination of the product navigation experiments.

Hills A. (2011). <u>MongoDB Journaling Performance – Single Server Durability.</u> Retrieved from:

http://www.adathedev.co.uk/2011/03/mongodb-journaling-performance-single.html.

Author and developer Adrian Hills discusses the journaling feature of MongoDB, and its effects

on overall performance. Essentially, Hills underscores the direct results of importing 295 megabytes of

data, with and without journaling enabled. This article was particularly helpful to this author when examining the ACID properties of MongoDB, to determine how much liability is posed by enabling journaling (to ensure durability).

Ingram E. (2012). <u>How MongoDB makes custom e-commerce easy.</u> Retrieved from: http://blog.mongodb.org/post/31729833608/how-mongodb-makes-custom-e-commerce-easy.

Forward Systems founder Eric Ingram discusses some of the advantages of using MongoDB as an e-commerce back-end database. Ingram illustrates that MongoDB's flexible schema and ability to quickly query on any field makes prototyping much faster in the early stages of development. He also cites the lack of database relations as a plus, due to the corresponding reduction in complexity. In answer to the question of atomic transactions, Ingram states that his implementation has not seen any related issues, and some alternate means to ensure data integrity.

Islam R. (2011). <u>PHP and MongoDB Web Development.</u> Packt Publishing Ltd. Birmingham, UK.

Author Rubayeet Islam covers some basic methods and use cases for accessing MongoDB from PHP. Islam begins by walking the reader through the basics of MongoDB as well as the construction of a very basic PHP blog application. The author also makes a point to describe basic HTTP connections as well as session management, and why it is important in the context of web application development. This source was heavily utilized during the early phases of this study. While not being referenced with great frequency, it did assist this author in prototyping early versions of the testing framework.

Kovacs K. (2012). <u>Cassandra vs. MongoDB vs. CouchDB vs. Redis vs. Riak vs. HBase vs.</u>

Membase vs. Neo4j comparison. **Retrieved from: http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis/.**

Author Kristof Kovacs compares and contrasts eight NoSQL databases of differing architectures. This article serves as a quick reference for those assessing NoSQL database architectures, to see which one will best fit their storage needs. While not necessarily an empirical source, this article was used early on in this study to evaluate different databases and assist the author in choosing MongoDB as the (NoSQL) focus for this study, as well as in creating the database assessment in Appendix H.

Kuhn D. (2010). Pro Oracle Database 11g Administration. Apress. New York, NY.

Author Darl Kuhn provides detailed insight into the tasks behind administering Oracle 11g. The author offers detailed descriptions on the installation and configuration process, as well as security, backups, and file management. Kuhn also points out several concepts and their real-world applications, and provides numerous SQL/Bash scripts to illustrate points of concept and assist with the automation of common tasks. The section on installation and configuration was central to building the Oracle database environment for this thesis.

Maddock J. (2003). Regular Expression Performance Comparison. Retrieved from:
http://www.boost.org/doc/libs/1_51_0/libs/regex/doc/gcc-performance.html.

Author and developer John Maddock, PhD compares the performance of the POSIX, PCRE and Boost regular expression libraries. As this reference is from a site designed to promote and distribute the Boost C++ libraries, its bias should be taken into consideration. However, it is relevant to this study in that it shows dramatic performance differences between PCRE and POSIX, and demonstrates that PCRE is almost on-par with Boost.

Membry P., Plugge E., Hawkins T. (2010). The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing. Springer-Verlag. New York, New York, NY.

Authors Peter Membry, Eelco Plugge and Tim Hawkins provide a reference for working with MongoDB that introduces the product and describes how to work with the data model. The authors provide examples on how to build MongoDB-based applications with PHP and Python. They also describe how to utilize GridFS to store large multimedia files with MongoDB, as well as other advanced topics such as backups and logging, as well as using the MongoDB profiler to analyze performance. This author found the chapter on replication helpful, and utilized it to get the MongoDB replication configuration running for this project.

Merriman D. (2012). M102 – MongoDB for DBAs. 10gen Education (online course material). Description retrieved from:

https://education.10gen.com/courses/10gen/M102/2013_Spring/about.

As a part of research gathering for this project, this author attended a 7-week online training course offered by 10gen, and taught by CEO Dwight Merriman. The course covered many aspects of MongoDB, including (but not limited to) configuration, replication, sharding, backup and recovery. Several points of information were gleaned from this course that provided this author with insight on testing MongoDB's adherence to ACID transaction properties. This course played an integral role in this author establishing adequate levels of understanding and comfort with MongoDB.

Mytton D. (2011). MongoDB vs. Cassandra. Retrieved from:

http://blog.serverdensity.com/mongodb-vs-cassandra/.

Author David Mytton compares various features of MongoDB with Cassandra, based on his experiences with each database. Mytton discusses everything from feature set to storage and replication strategies of each, as well as levels of support available. One key point the author makes, is

in the configuration of consistency levels as he notes that Cassandra offers more levels of granularity.

One of Mytton's conclusions states that product maturity must play a role in the decision process, and

cites 10gen's support for MongoDB as "well worth the money."

Nanda A. (2008). Oracle Database 11g: The Top Features for DBAs and Developers – Data

Warehousing and OLAP. Oracle Corporation. Retrieved from:

http://www.oracle.com/technetwork/articles/sql/11g-dw-olap-100058.html.

DBA and Author Arup Nanda describes the variety of data warehousing tools available with

Oracle 11g. Nanda discusses the details behind Cube-Organized Materialized Views, Query Rewrite,

Partition Change Tracking with materialized views, the Analytic Workspace Manager and other

business intelligence based features. This article is significant due to highlighting one of the benefits of

Oracle 11g, as most enterprise organizations have a need for business intelligence tools.

Oliver A. (2012). Ill-informed Haters Go After MongoDB. InfoWorld. Retrieved from:

http://www.infoworld.com/d/application-development/ill-informed-haters-go-after-mongodb-

205096?page=0,1.

InfoWorld columnist Andrew Oliver discusses several instances of unsatisfied individuals who

migrated away from MongoDB. After analyzing each instance of complaint, Oliver illustrates the

technical issues behind why MongoDB was not the correct solution in each case. The author states

several times that picking a technology based on current trends is just not a good idea. This article is

significant because it helps identify much of the general lack of understanding of NoSQL technology

(MongoDB in particular), as well as some of the more unfortunate things that can occur when a

technology is chosen for the wrong reasons.

PCI Security Standards Council (2010). PCI DSS Quick Reference Guide. Retrieved from:

https://www.pcisecuritystandards.org/documents/PCI%20SSC%20Quick%20Reference%20Gui

de.pdf.

The PCI DSS Quick Reference guide is provided by the PCI (Payment Card Industry) Security

Standards Council provides a quick look-up guide to be used by those designing and securing a

website. Aspects of these standards apply to many aspects of the website including (but not limited to)

site development techniques, network security and business processes. This guide was briefly

referenced in this study to illustrate the importance of payment card data encryption, even though it is

beyond the scope of this study.

Ries. S. (2008). <u>Head To Head: A Comparison Between Windows and Linux as an Oracle</u>

<u>Database Platform.</u> CSC North American Public Sector. Retrieved from:

http://assets1.csc.com/lef/downloads/Windows_Linux_OracleDP.pdf.

Author and researcher Steve Ries discusses the choice between Windows or Linux to run

production versions of Oracle database software. The author installed Oracle on both Windows and

Linux on matching hardware, and then conducted a set of experiments testing throughput and resource

usage during various Oracle DB operations. The author's conclusions show overwhelming support for

Linux. This work is significant, because it helps to substantiate this author's choice in using Linux as a

database server platform for these experiments.

Tindel C. (2012). <u>Intro to Schema Design.</u> MongoDB – Chicago (lecture/presentation). 10gen.

Chicago Hyatt Magnificent Mile, Chicago, IL. Retrieved from:

http://www.10gen.com/presentations/mongodb-chicago-2012/intro-schema-design.

10gen Solution Architect Chad Tindel discusses schema design for MongoDB. His presentation

was aimed at audiences familiar with developing schema for RDBMSs, and specifically how these two

approaches differ. Tindel drives a main point where the focus in RDBMS schema design is on data

storage, while MongoDB's de-normalized approach focuses on data use. In his presentation, Tindel

shows how to model common RDBMS-based problems of foreign key relations, (tree) hierarchy

modeling, as well as embedding documents in arrays to solve many-to-many relationships.

Trelle T. (2013). <u>MongoDB Text Search Explained.</u> codecentric AG. Retrieved from:

http://blog.codecentric.de/en/2013/01/text-search-mongodb-stemming/.

Author Tobias Trelle examines MongoDB's text search capability released with MongoDB 2.4

(for development). Trelle discusses a little about how full text searching typically functions, and

discusses common search topics such as stop words and stemming. This article is relevant, because

this study demonstrates the regular expression search capabilities of MongoDB, which further

underscores 10gen's intent to take implementation of this feature seriously.

Yu H., Vahdat A. (2000). <u>Design and Evaluation of a Continuous Consistency Model for</u>

<u>Replicated Services.</u> Department of Computer Science – Duke University. Durham, NC.

This paper by authors Haifeng Yu and Amin Vahdat discusses the advantages of relaxing

consistency requirements in distributed systems. They provide results from experiments designed to

measure performance, consistency and data staleness. Yu and Vahdat suggest a middleware design to

support "optimistic consistency" and show the performance benefits attained using their middleware in

their experiments. This work is significant to this thesis, because it provides a detailed analysis of the

tradeoffs involved in attaining optimal performance in a distributed system.

Appendix C: Oracle and MongoDB Comparison of Terms and Keywords

Oracle	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
Primary Key	Primary Key
GROUP BY	Aggregation Framework
JOIN	n/a (documents can be embedded)

Table C-1. A comparison of Oracle and MongoDB database terms/functionality.

Oracle	MongoDB
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM	$sum
COUNT	$sum

Table C-2. A comparison of Oracle and MongoDB aggregation operators (10gen, 2012).

Appendix D: E-CommerceDB Schema (Oracle)

The schema for the Oracle 11g R2 "EcommDB" database was mainly designed around the "Products," "Customers," and "Orders" entities. The tables and their relationships can be seen in Figure D-1. Indexes for the tables are shown in table D-1. While not shown in the ER diagram, this database also contains other database objects, such as foreign key constraints, sequences, and triggers. These can be seen in the DDL code shown in Figures D-2, D-3, D-4, and D-5.
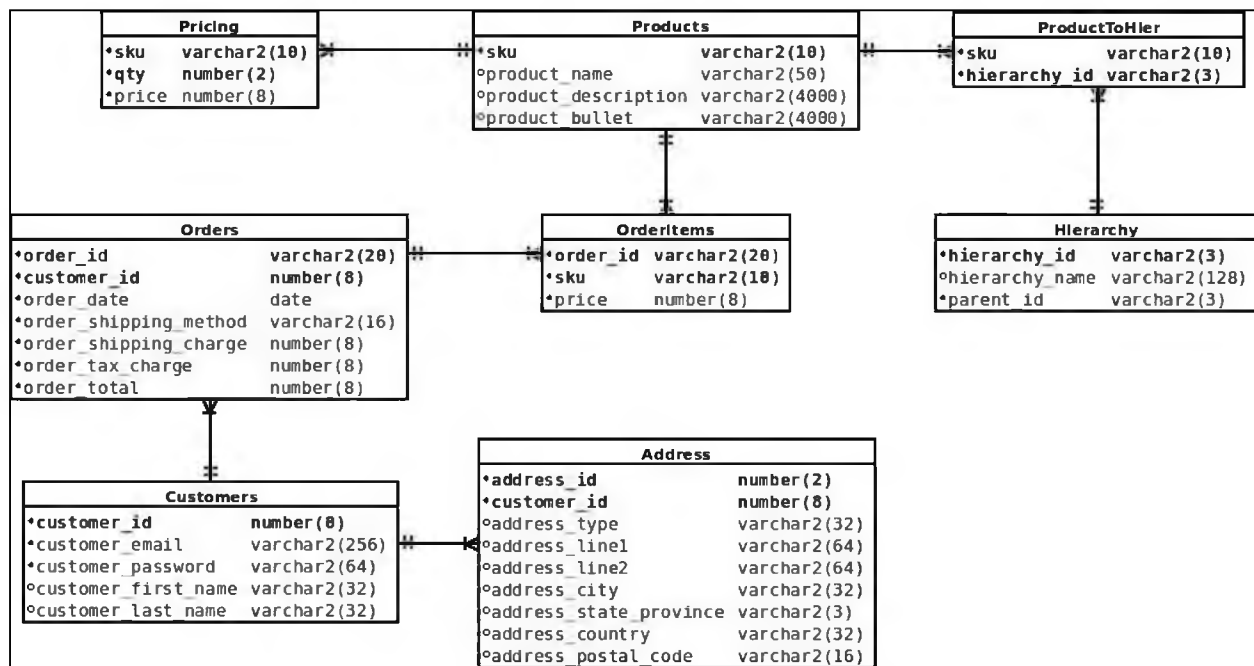
Figure D-1. Oracle database e-commerce ER diagram.

The products table (DDL in Figure D-2) was determined to have a natural key for the stock keeping unit (SKU). SKU would be the both the primary key and lone indexed field on this table. While there could be several additional product attributes used in e-commerce, only the product's name, description, and bulleted list of features were added to the table. These fields were included because they were determined to be the most-challenging to search through, and thus provide a better view of search capabilities.

INDEX_NAME	COLUMN_NAME	TABLE_NAME	COLUMN_POSITION
PK_ADDRESS	ADDRESS_ID	ADDRESS	1
PK_ADDRESS	CUSTOMER_ID	ADDRESS	2
CUSTOMER_EMAIL_IDX	CUSTOMER_EMAIL	CUSTOMERS	1
PK_CUSTOMERS	CUSTOMER_ID	CUSTOMERS	1
HIERARCHY_IDX2	PARENT_ID	HIERARCHY	1
PK_HIERARCHY	HIERARCHY_ID	HIERARCHY	1
PK_ORDERITEMS	ORDER_ID	ORDERITEMS	1
PK_ORDERITEMS	LINE_ITEM_ID	ORDERITEMS	2
PK1_ORDERS	ORDER_ID	ORDERS	1
PK_PRICING	SKU	PRICING	1
PK_PRICING	PRICE_QTY	PRICING	2
PK_PRODUCTS	SKU	PRODUCTS	1
PK_PRODUCTTOHIER	SKU	PRODUCTTOHIER	1
PK_PRODUCTTOHIER	HIERARCHY_ID	PRODUCTTOHIER	2

Table D-1. Indexes used in the Oracle 11g R2 database.

The pricing table (Figure D-2) was created to hold both quantity and price for a SKU. This

table was split-out on its own, as one product (SKU) can have many prices. This is because many e-

commerce retailers will provide discounts to customers who buy multiple quantities at once. The

pricing table utilizes a concatenated key, consisting of both the SKU and quantity.

The hierarchy table (Figure D-2) was created to keep track of which categories the products belonged to. The hierarchy table contained only the hierarchy id (primary key), name, and parent id. Parent id was also given an index, as it was used to query parent hierarchies from the same table. The products and hierarchy table relationships were maintained by the "ProductToHier" bridge table. This table was necessary because categories have multiple products, and some products can belong to more than one category.

```
CREATE TABLE Hierarchy(

hierarchy_id VARCHAR2(3),

hierarchy_name VARCHAR2(128),

parent_id VARCHAR2(3),

CONSTRAINT pk_hierarchy PRIMARY KEY (hierarchy_id));


CREATE TABLE Products(

sku VARCHAR2(16),

product_name VARCHAR2(128),

product_description VARCHAR2(4000),

product_bullet VARCHAR2(4000),

CONSTRAINT pk_products PRIMARY KEY (sku);


CREATE TABLE ProductToHier(

sku VARCHAR2(16),

hierarchy_id VARCHAR2(3),

CONSTRAINT pk_productToHier PRIMARY KEY (sku,hierarchy_id),

CONSTRAINT fk1_productToHier FOREIGN KEY (sku) REFERENCES Products (sku),

CONSTRAINT fk2_productToHier FOREIGN KEY (hierarchy_id) REFERENCES Hierarchy
(hierarchy_id));


CREATE TABLE Pricing(

sku VARCHAR2(16),

price_qty NUMBER(2),

price NUMBER(6,2),

CONSTRAINT pk_pricing PRIMARY KEY (sku, price_qty),

CONSTRAINT fk1_pricing FOREIGN KEY (sku) REFERENCES Products (sku));
```

Figure D-2. DDL for Oracle e-commerce hierarchy, products, productToHier, and pricing tables.

The customers table (Figure D-3) contains various data about our registered customers, including first and last names, email address, and password. It has a primary key on the customer id. The customer's email address is also indexed, as the customer records are queried by the application

based on that field. The uniqueness of the customer id is maintained by a sequence which is

maintained on the database and incremented by a trigger that fires on an insert operation.

```
CREATE SEQUENCE customer_id_seq

INCREMENT BY 1

START WITH 3000000

NOCYCLE NOCACHE;


CREATE TABLE Customers(

customer_id NUMBER(8),

customer_email VARCHAR2(256),

customer_password VARCHAR2(64),

customer_first_name VARCHAR2(32),

customer_last_name VARCHAR2(32),

CONSTRAINT pk_customers PRIMARY KEY (customer_id));


CREATE OR REPLACE TRIGGER customer_iid

BEFORE INSERT ON customers

FOR EACH ROW

BEGIN

  SELECT customer_id_seq.NEXTVAL

  INTO    :new.customer_id

  FROM    dual;

END;
```

Figure D-3. DDL for Oracle e-commerce customer table, including the trigger and sequence.

The orders table (Figure D-4) was designed to keep track of each order placed on the website.

This table uses the application-generated order_id as its primary key. The customer id is a foreign key

which references the customers table. The remaining fields are generally considered to be vital order

data: date, shipping details, taxes, and total amount due. The orders and products table relationships

are maintained by the "OrderItems" bridge table. This table is necessary because products can be

ordered many times, and orders contain many products.

```
CREATE TABLE OrderItems(

order_id VARCHAR2(20),

line_item_id NUMBER(3),

sku VARCHAR2(16),

qty NUMBER(5),

CONSTRAINT pk_orderitems PRIMARY KEY (order_id, line_item_id),

CONSTRAINT fk1_orderitems FOREIGN KEY (sku) REFERENCES Products (sku));


CREATE TABLE Orders(

order_id VARCHAR2(20) NOT NULL,

customer_id NUMBER(8),

order_date DATE,

order_ship_method VARCHAR2(16),

order_ship_charge NUMBER(8),

order_tax_charge NUMBER(8),

order_total NUMBER(8),

CONSTRAINT pk1_orders PRIMARY KEY (order_id),

CONSTRAINT fk1_orders FOREIGN KEY (customer_id) REFERENCES Customers
(customer_id));
```

Figure D-4. DDL for Oracle e-commerce tables order and orderItems.

The address table (Figure D-5) contains typical data describing our customer's postal mailing

addresses. It maintains its uniqueness with a concatenated key made up of a two-digit value from the

application and the eight-digit customer id. The address entity is tied to the customer table in a many-

to-one relationship, as one customer may have multiple addresses (shipping, billing, etc…).

```
CREATE TABLE Address(

address_id NUMBER(2),

customer_id NUMBER(8),

address_type VARCHAR2(32),

address_line1 VARCHAR2(64),

address_line2 VARCHAR2(64),

address_city VARCHAR2(32),

address_state_province VARCHAR2(32),

address_country VARCHAR(32),

address_postal_code VARCHAR(16),

CONSTRAINT pk_address PRIMARY KEY  (address_id, customer_id),

CONSTRAINT fk1_address FOREIGN KEY  (customer_id) REFERENCES Customers
(customer_id));
```

Figure D-5. DDL for Oracle e-commerce address table.

Appendix E: E-commerceDB Schema (MongoDB)

Like its Oracle 11g R2 counterpart, the MongoDB "ecommerceDB" database was mainly

designed around the "products," "customers" and "orders" entities. These collections (shown in Figure

E-1) were not related in any way at the database level, and duplicated some data to increase

performance. MongoDB databases and collections are not required to be created before use, so there is

no equivalent SQL DDL code to show.

```
orders                          customers                       products
{                               {                               {
    _id,                            _id,                            _id,
    customer_id,                    email,                          sku,
    order_date,                     password,                       product_name,
    order_ship_method,              firstname,                      product_description,
    order_ship_charge,              lastname,                       product_bullet,
    order_tax_charge,               addresses: [                    category_name,
    order_total,                        (                           hierarchy_name,
    order_items: [                          _id,                    pricing: [
        (                                   type,                       (
            _id,                            line1,                          qty,
            sku,                            line2,                          price
            qty,                            city,                       )
            price                           state_province,             ]
        )                                   postal,                 }
        ]                                   country
}                                       )
                                        ]
                                }
```

Figure E-1. MongoDB database e-commerce schema.

One collection not pictured above is the "categories" collection. It was created during the

product navigation experiments in a [successful] attempt to rectify a performance issue (Figure E-2).

Essentially, the hierarchy_name and category_name properties were split-off from the "products"

collection. A query of all categories for a specific hierarchy_name, would also return the number of

SKUs and a list of all SKUs under each category. While the number of SKUs is something that could

be computed at query-time, it was determined to be faster to store that value rather than compute it.

```
categories
{
        _id,
        hierarchy,
        category,
        skuList,
        numSKUs

}
```

Figure E-2. Categories collection, broken-off from the products collection, and drastically increased product navigation performance.

Embedded documents and some data redundancy was necessary in this model due to the fact that there is no equivalent of a SQL "join" in MongoDB. Duplicated data can be seen in both the "categories" and "products" collections (the category property). Embedded documents can be seen in the "customers," "orders," and "products" collections. For instance, customers require the ability to have multiple addresses, products can have multiple prices, and orders contain multiple products.

Appendix F: Data Generation and Insert Code

This project required a lot of data about many different entities. To maximize the accuracy of my results, I would need data for customers, addresses, and products. The customer data was compiled by creating several text files filled with assorted names for customers, streets, and cities (shown in Figure F-1) and loaded into local Array List objects.

```
FileWriter fwOut = new
FileWriter("/home/aploetz/programming/java/thesis_data/customerData.txt");

ArrayList<String> firstNames =
getNames("/home/aploetz/programming/java/thesis_data/firstNames.txt");

ArrayList<String> lastNames =
getNames("/home/aploetz/programming/java/thesis_data/lastNames.txt");

ArrayList<String> streetNames =
getNames("/home/aploetz/programming/java/thesis_data/streetNames.txt");

ArrayList<City> cities =
getCities("/home/aploetz/programming/java/thesis_data/cityNames.txt");

ArrayList<String> streetTypes =
getNames("/home/aploetz/programming/java/thesis_data/streetTypes.txt");

ArrayList<String> emailProviders =
getNames("/home/aploetz/programming/java/thesis_data/emailProviders.txt");
```

Figure F-1. Java code to load array lists with data from the prepared files.

These array lists were then accessed with a random index to build data for each customer (Figure F-2). Once the customer's name and email were compiled, a random number of customer addresses was generated. These addresses were then added to the customer name data, and written to

the pipe-delimited file.

```
for (int intCounterJ = 0; intCounterJ < intCustomerCount; intCounterJ++) {

        intFirstNameIndex = randomNumber(0,firstNames.size() - 1);
        intLastNameIndex = randomNumber(0,lastNames.size() - 1);
        while (firstNames.get(intFirstNameIndex) ==
lastNames.get(intLastNameIndex)) {
                intLastNameIndex = randomNumber(0,lastNames.size() - 1);
        }

        strFirstName = firstNames.get(intFirstNameIndex);
        strLastName = lastNames.get(intLastNameIndex);
        strEmail = genEmail(strFirstName,strLastName,emailProviders);
        strPassword = genPassword();
        //gen one or two addresses
        intNumOfAddresses = randomNumber(1,2);

        for (int intCounterK = 0; intCounterK < intNumOfAddresses; intCounterK++) {
                intStreetNameIndex = randomNumber(0,streetNames.size() - 1);
                intCityIndex = randomNumber(0,cities.size() - 1);
                intStreetTypeIndex = randomNumber(0,streetTypes.size() - 1);
                intAddressLine2 = randomNumber(0,5);
                intStreetNumber = randomNumber(100,99999);
                strAddress = "";
                strAddress += Integer.toString(intStreetNumber);
                strAddress += " " + streetNames.get(intStreetNameIndex);
                strAddress += " " + streetTypes.get(intStreetTypeIndex);

                if (intAddressLine2 == 4) {
                  strAddress += " Apt " + Integer.toString(randomNumber(0,9999));
                } else if (intAddressLine2 == 5) {
                  strAddress += " Ste " + Integer.toString(randomNumber(0,9999));
                }

                strAddress += "|" + cities.get(intCityIndex).getCity();
                strAddress += "|" + cities.get(intCityIndex).getState();
                strAddress += "|" + cities.get(intCityIndex).getZip();
                strAddress += "|" + cities.get(intCityIndex).getCountry();
                //write address to file
                fwOut.flush();
                fwOut.write(strFirstName + "|" + strLastName + "|" + strEmail + "|"
+ strPassword + "|" + strAddress + "\r\n");
        }
}
```

Figure F-2. Java code to piece-together the customer data file from random parts.

The end result was a single, pipe-delimited file of customer data which was then loaded into

both MongoDB and Oracle. For Oracle, the customer data was handled first (Figure F-3). The code to

insert customer data into MongoDB can be seen in Figure F-4.

```
String strSQL = "INSERT INTO customers
(customer_email,customer_password,customer_first_name,customer_last_name) ";
strSQL += "VALUES(?,?,?,?) ";

//get customer search keys (firstname, lastname)
String firstname = custData.getFirstName();
String lastname = custData.getLastName();
String email = custData.getEmail();

intCustomerID = getCustomerIDFromOracle(oConn, email);

if (intCustomerID == 0) {
//INSERT customer data into customers table
try {
        stmt = oConn.prepareStatement(strSQL);
        stmt.setString(1, email);
        stmt.setString(2, custData.getPassword());
        stmt.setString(3, firstname);
        stmt.setString(4, lastname);
        stmt.execute();
        stmt.close();
        oConn.commit();
} catch (SQLException se) {
        System.out.println(se.toString());
}
```

Figure F-3. Java code to insert customer data into Oracle.

```
newCustomer = new BasicDBObject();

newCustomer.put("firstname", firstname);

newCustomer.put("lastname", lastname);

newCustomer.put("email", email);

newCustomer.put("password", custData.getPassword());


if (addressData.getType() != null) {

    newAddress.put("type", addressData.getType());

}


newAddress.put("line1", addressData.getLine1());


if (addressData.getLine2() != null) {

    newAddress.put("line2", addressData.getLine2());

}


newAddress.put("city", tempCity.getCity());

newAddress.put("state", tempCity.getState());

newAddress.put("postal", tempCity.getZip());

newAddress.put("country", tempCity.getCountry());


addressList.add(newAddress);

newCustomer.put("address", addressList);


customerCollection.insert(newCustomer);
```

Figure F-4. Java code to insert customer data into MongoDB.

Please note that the code shown was formatted to best-fit the page. During this formatting

process, some spacing and comments were eliminated. The complete source code can be viewed at:

https://github.com/aploetz/ploetzThesisCode/tree/master/DataTools/src/www/aaronstechcenter/com/dat

atools/DataTools.java.

Appendix G: Testing Framework Code

This appendix will present some of the testing framework code, highlighting the Oracle and MongoDB access methods used for the customer update experiment. The code to call Oracle can be seen in Figure G-1. Note that the customer_password and customer_email fields (in the SET and WHERE clauses, respectively) are shown to be set/compared to question marks. This indicates that prepared statements were used, and the parameters were provided later on in the code.

```
    Connection conn = getOracleConnection();
String strSQL = "UPDATE customers " +
    "SET customer_password = ? " +
    "WHERE customer_email = ? ";
for (String email : customers_) {
    try {
            String strPassword = genPassword();
            PreparedStatement stmt = conn.prepareStatement(strSQL);
            stmt.setString(1, strPassword);
            long beginDate = System.nanoTime();
            stmt.executeUpdate();
            conn.commit();
            stmt.close();
            long endDate = System.nanoTime();
            TLog localLog = new TLog(beginDate, endDate, strSQL);
            log_.add(localLog);
    } catch (SQLException ex) {
            System.out.println(ex);
    }
}
        //all done, close connection
conn.close();
```

Figure G-1. Java code to simulate an update to a group of customer records stored in Oracle.

The code to call MongoDB can be seen in Figure G-2. As the Java API for MongoDB does not use a SQL-like query language, the equivalent MongoDB query is compiled and logged at the end.

```
Mongo mConn = getMongoDBConnection();

DB mDB = mConn.getDB("ecommerceDB");

DBCollection customerCollection = mDB.getCollection("customers");


for (String email : customerEmails_) {

        String strPassword = genPassword();

        BasicDBObject custMod = new BasicDBObject();

        custMod.put("email", email);

        custMod.put("password", strPassword);

        //set beginTime

        long beginDate = System.nanoTime();

        //update the customer

        customerCollection.update(new BasicDBObject().append("email", email),
custMod);

        //set endDate

        long endDate = System.nanoTime();

        TLog localLog = new TLog(beginDate, endDate,
"db.customers.update({email: \"" + email + "\"},{password: \"" + strPassword +
"\"})");

        log_.add(localLog);

    }

    mConn.close();
```

Figure G-2. Java code to simulate an update to a group of customer records stored in MongoDB.

The SQL for the product navigation experiments can be seen in Figures G-3, G-4, and G-5. Similar to the code shown in Figure G-1, the SQL shown in these Figures (G-3, G-4, and G-5) is also parameterized, as their variable values were passed via prepared statements.

```
SELECT h.hierarchy_id

FROM hierarchy h

WHERE h.parent_id = ?

AND (SELECT COUNT(*)

     FROM producttohier p2h

     WHERE p2h.hierarchy_id = h.hierarchy_id) > 0)
```

Figure G-3. SQL code for the top tier product navigation in Oracle.

```
SELECT SKU

FROM producttohier p

WHERE p.hierarchy_id = ?
```

Figure G-4. SQL code for product group navigation in Oracle.

```
SELECT *

FROM products

WHERE SKU = ?
```

Figure G-5. SQL code for single product navigation in Oracle.

The SQL for the product searches is listed in Figures G-6 and G-7.

```
SELECT *

FROM products

WHERE regexp_like(product_description, ?, 'c')
```

Figure G-6. SQL code for case-sensitive product searching in Oracle.

```
SELECT *

FROM products

WHERE regexp_like(product_description, ?, 'i')
```

Figure G-7. SQL code for case-insensitive product searching in Oracle.

The SQL for the customer update was shown in Figure G-1, but is also shown in Figure G-8. Here, the query stands out on its own, instead of having to be located inside the code block shown in Figure G-1.

```
UPDATE customers
SET customer_password = ?
WHERE customer_email = ?
```

Figure G-8. SQL code for customer updates in Oracle.

The order placement SQL can be seen in Figures G-9, G-10, and G-11. As mentioned in the descriptions of previous queries, these Figures utilize parameterized queries for prepared statements (as shown by the question marks). For the price lookup (Figure G-9), rows were returned based-on the SKU number and the desired quantity break. The rows returned would be put in descending order, and only the first row would be returned. This would ensure that the returned row would be for the desired quantity break.

```
SELECT price
FROM pricing
WHERE SKU = ? and price_qty <= ? and ROWNUM = 1
ORDER BY price_qty DESC
```

Figure G-9. SQL code to perform a price lookup for the order process in Oracle.

The order insert code (Figure G-10) inserts the data which is unique to that particular order. Again, this query is parameterized with all entries in the VALUES clause being question marks, indicating that the specific values to be inserted were submitted via a prepared statement.

```
INSERT INTO orders(order_id, customer_id, order_date, order_ship_method,
order_ship_charge, order_tax_charge, order_total)
VALUES(?,?,?,?,?,?,?)
```

Figure G-10. SQL code to insert order records into Oracle.

The order items insert code (Figure G-11) took care of inserting each specific item required for the order. This data would typically be retrieved with a SQL join with the order table, as orderitems

shares a many-to-one relationship with the orders table.

```
INSERT INTO orderitems(order_id, line_item_id, sku, qty)
VALUES(?,?,?,?)
```

Figure G-11. SQL code to insert order items into Oracle.

And finally, the price update SQL is shown in Figure G-12. Note that the price and SKU fields (in the SET and WHERE clauses, respectively) are shown to be set/compared to question marks. This indicates that prepared statements were used, and the parameters were provided later on in the code. As the quantity break (price_qty) was a randomly created variable, the experiments were run to always update the price where qty_break was equal to one. This is because each row in the pricing table would start with (and therefore always have) a quantity break for one.

```
UPDATE pricing
SET price = ?
WHERE SKU = ?
AND price_qty = 1
```

Figure G-12. SQL code to update product prices in Oracle.

Note that this appendix focused largely on the SQL behind each piece of functionality in the testing framework, and much of that focus was on the Oracle side. As the MongoDB Java API does not use a SQL-like query language, similar queries for the MongoDB functions were not shown. The complete code (for both the Oracle and MongoDB functions) can be viewed at the address mentioned below.

Please note that the code shown was formatted to best-fit the page. During this formatting process, some spacing and comments were eliminated. The complete source code can be viewed at: https://github.com/aploetz/ploetzThesisCode/blob/master/ThesisFramework/src/www/aaronstechcenter/com/thesis/ThesisFramework.java

Appendix H: Database Architecture Questionnaire

Considering an upcoming or current application of yours, please rate each of the following requirements by selecting an answer in either column A, B, C, or D.

		A	B	C	D
1	Data security		Very Important		Not Important
2	Ease of DB installation		Not Important		Very Important
3	Flexible schema		Not Important		Very Important
4	Ad-hoc queries		Very Important		Not Important
5	Built-in analytics tools		Very Important		Not Important
6	Geospatial indexing	Somewhat Important	Not Important	Very Important	
7	Map/reduce		Not Important		Very Important
8	Full text search	Not Important	Very Important	Somewhat Important	

9	Fault tolerance	Very Important	Not Important	Somewhat Important	
10	More reads than writes	Somewhat Important	Not Important	Very Important	
11	More writes than reads	Very Important	Not Important	Somewhat Important	
12	Transaction atomicity		Very Important		Not Important
13	Data consistency	Not Important	Somewhat Important	Very Important	
14	High-availability	Very Important	Somewhat Important	Not Important	
15	Ease of horizontal scaling		Not Important		Very Important
16	Large data performance		Not Important		Very Important
17	Relational data support		Very important		Not important

Now total your scores for each column. If your score:

- For column A was 4 or more, you should consider a highly-available, partition-tolerant NoSQL database (Cassandra, Riak, Dynamo).

- For column B was 9 or more, you should consider a RDBMS (Oracle, MSSQL, MySQL).

- For column C was 4 or more, you should consider a strong-consistent, partition-tolerant NoSQL database (MongoDB, HBase, BigTable).

- For column D was 6 or more, you should consider a NoSQL database, but further analysis of your requirements is necessary to choose the correct one.

Please note that these questions and answers are based on the general qualities of each type of database architecture. For instance, most NoSQL databases that I have worked with are relatively easy to install when compared to their relational counterparts; hence the "Very Important" option in the NoSQL database column for "Ease of installation." Now that is not to say that all NoSQL databases are easy to install, but rather that "generally-speaking" NoSQL databases tend to be easier to install than relational databases.

It is entirely possible to achieve qualifying scores in more than one column. It is also possible to not register a qualifying score for any of the columns. This questionnaire is not definitive, but is intended to be used as an aid in exploring database technologies that may be appropriate for your application.