Bachelorarbeit

Ngoc Huyen Nguyen

An application-oriented comparison of two NoSQL database systems: MongoDB and VoltDB

Ngoc Huyen Nguyen

An application-oriented comparison of two NoSQL database systems: MongoDB and VoltDB

Bachelorarbeit eingereicht im Rahmen der BachelorprÃ¼fung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Wolfgang Gerken

Eingereicht am: 8. August 2016

Ngoc Huyen Nguyen

Thema der Arbeit

An application-oriented comparison of two NoSQL database systems: MongoDB and VoltDB

Stichworte

Big Data, NoSQL, SQL, application-oriented comparison, MongodB, VoltdB

Kurzzusammenfassung

Diese Bachelorarbeit stellt ein anwendung-orientierter Vergleich zwischen MongoDB, ein NoSQL Vertreter, und VoltDB, ein NewSQL Vertreter, an. Es gibt zwei Testszenarien: ecommerce und social network. Die Tests werden nach definierten Kriterien und Testszenarien aufgebaut und dient dazu, die Leistung jeder Testdatenbank in jedem Testszenario zu evaluieren.

Ngoc Huyen Nguyen

Title of the paper

An application-oriented comparison of two NoSQL database systems: MongoDB and VoltDB

Keywords

Big Data, NoSQL, SQL, application-oriented comparison, MongodB, VoltdB

Abstract

This thesis conducts an application-oriented comparison of two databases MongoDB, a NoSQL representative, and VoltDB, a NewSQL representative. There are two test scenarios: ecommerce and social network. The test queries are tailored by defined test criteria and test scenario to investigate the performance of each database in each test scenario.

Contents

Listings

1 Introduction

1.1 Motivation

Big Data has become common over the years. With the exponential growth of data every year, storing data in standalone systems is no longer possible. Databases should be able to deploy in a cluster to spread the workloads, increase overall performance and availability.

Every year, there are more and more database products coming to the market. They belong either to SQL or NoSQL database group. Since their appearance, NoSQL databases has dominated in handling Big Data. Traditional SQL databases, which lack horizontal scaling support, can not compete with NoSQL in providing a system with high availability and fast performance.

1.2 Goals

MongoDB, as one of the most popular NoSQL database, has been frequently updated with new features. One of them is the ability to join tables, which reduces the distance between SQL and NoSQL databases.

The introduction of NewSQL creates another horizon for SQL databases. Keeping the relational data model and ACID transaction support, which ensures data consistency and system durability, there is possibility to scale the whole system horizontally. New features which support distributing environment, such as single-partitioned transaction, are also introduced in the NewSQL database. One prominent representative of this group is VoltDB. Claimed as a hybrid solution that can fulfill the needs for data consistency, reliability in SQL databases and data sharding, higher performance and availability in NoSQL databases

The goal of this thesis is to put both databases in scenarios, which are typical either for NoSQL or SQL databases in order to investigate how close they are to replace the databases in the other group, their strength and their weakness, which one is more suitable for which scenario.

2 Basics

2.1 Big Data

2.1.1 Definition

Big Data is a term used to indicate large and complex data sets that require special technologies to store and process.

In the last few decades, data has grown exponentially and shown no signs to stop. According to IDC's report Gantz und Reinsel (2010) in 2011, the digital universe set a record in 2010 by growing by 62% to nearly 800,000 petabytes. It was also predicted that the digital universe would be 44 times as big as it was in 2009, which was approximately 35 zetabytes.

Although storing a large amount of data for analytic and predictive purposes has existed for a long time, the first document that has ever used this term is a paper written by the scientists in NASA in 1997 Cox und Ellsworth (1997).

In February 2001, Doug Laney, an analyst of Meta Group, used three features to summarize Big Data in his paper Laney (2001). Those features are the 3Vs:

Velocity Data can be accessed and streamed in a short time, ideally near real-time. The latency of read and write operations should be as short as possible.

Volume This involves very large datasets, ranging from tens of terabytes to hundreds of petabytes.

Variety Data can exist in various formats - from structured, semi-structured to unstructured data likes video, pictures, texts, emails,..

Since then the "3Vs" has been used by anyone who attempts to describe "Big Data". Other companies and individuals sometimes have additional features to define big data. For example, Oracle defines another fourth "V" beside the predifined "3Vs" as Value to indicate the intrinsic value that can be derived from data.

2.1.2 Current status

Nowadays Big Data has become more and more popular and has been applied in many different fields - from business, healthcare, climate science to social medias.

In business, data acquisition and analysis is essential to gain more insights which can boost business performance. Companies can tap into social media sources like Facebook, Twitter to conduct sentimental analysis, mine comments, compliments and complaints from the customers about their products or to understand current trends. Based on these analyses, changes will then be made to improve performance. Big Data can also be used for fraud detection. Fraud detections used to be carried out every few months when it's already too late to diminish the damage. By using big data analytics to detect fraudulent behaviors in enormous datasets from multiple sources, insurance companies can now detect fraud in real time.

Big Data has gradually changed health care's horizon. One example is Google Flu Trends, which attempts to predict flu outbreaks based on the numbers of key search term. According to BBC in 2014 Wall (2014), analysis of mobile phone data helped predict the spread of Ebola virus by coming up with detailed maps of population movements. In a report by Mc Kinsey Institute in 2013 Basel Kayyali und Kuiken (2013), it is estimated to save up to $450 billion in health-care spending in the USA by using Big Data applications on a large scale.

According to Gunelius (2014), "five exabytes of content were created between the birth of the world and 2003. In 2013, 5 exabytes of content were created each day. " With the data explosion in social medias "*Every minute: Facebook users share nearly 2.5 million pieces of content, Twitter users tweet nearly 300,000 times. Instagram users post nearly 220,000 new photos. YouTube users upload 72 hours of new video content. Apple users download nearly 50,000 apps. Email users send over 200 million messages." - Susan Gunelius* the needs of systems and mechanisms that can handle such enormous amount of data has also grown stronger than ever.

Big Data is not a new concept. In fact, it has existed for decades. However, with the data explosion of the Internet era together with the introduction of many new technologies every year, it is guaranteed that Big Data will become more and more popular in the future.

2.2 A general comparison between SQL and NoSQL

2.2.1 Transaction and consistency models

While SQL databases all support transactions, most NoSQL database don't have this feature or only support transactions partly.
The two most common consistency models among databases are **ACID** and **BASE**:

ACID

Traditional relational databases place great emphasis on keeping the data consistent. They achieve this by implementing **ACID** transactions. **ACID** stands for **A**tomicity, **C**onsistenty, **I**solation, **D**urability:

Atomicity The transaction can either be carried out completely or not at all. If one part of the transaction fails, the whole transaction fails and the database stays unchanged.

Consistency Any transaction will bring the database from one valid state to another valid state. That means no violation of predefined rules, e.g., constraints, is allowed.

Isolation Transactions are independent of each other. Transactions which are executed together or serially will result in the same state of the system.

Durability After a transaction is committed, the resulting state of the system will remain so even in case of system failure, power loss or other types of system breakdowns.

BASE

Though ACID principle ensures the consistency of data at all times, there are cases then it is pessimistic and has negative impact on performance aspects of the whole system, like scalability and flexibility. BASE, on the other hand, is optimistic and more relaxed on the consistency front.

BASE is the abbreviation for:

Basic avalability The database is accessible most of the time. This availability can be achieved through partial failure support. For example, for a system that is scaled out on many servers, when one of them fails, only the users on this server get affected. The rest of the system stays alive and functions properly.

Soft state The system always stays in a "soft" state. Even when there are no new inputs, the system can still undergo changes due to "eventual consistency".

Eventual consistency The system continues to receive inputs and will "eventually" be consistent when there are no new inputs. The system doesn't force or check consistency after each transaction. Transactions can be conducted immediately or postponed till later.

CAP Theorem

CAP Theorem, also called Brewer Theorem, was presented by Eric Brewer in 2002 at the ACM Symposium on the Principles of Distributed Computing. The theorem focuses on describing the three necessary requirements for successful design and implementation of distributed computer systems. **CAP** stands for:

Consistency Updating one copy of the data on one node will result in updating all copies of the data.

Availability Every node of the system should always respond to received queries unless it dies.

Partition Tolerance The whole system continues to operate even if partitions or the networks between partitions fail.

According to CAP Theorem, it is impossible to meet all three requirements of the theorem. Figure 2.1 shows the CAP Theorem Diagram:

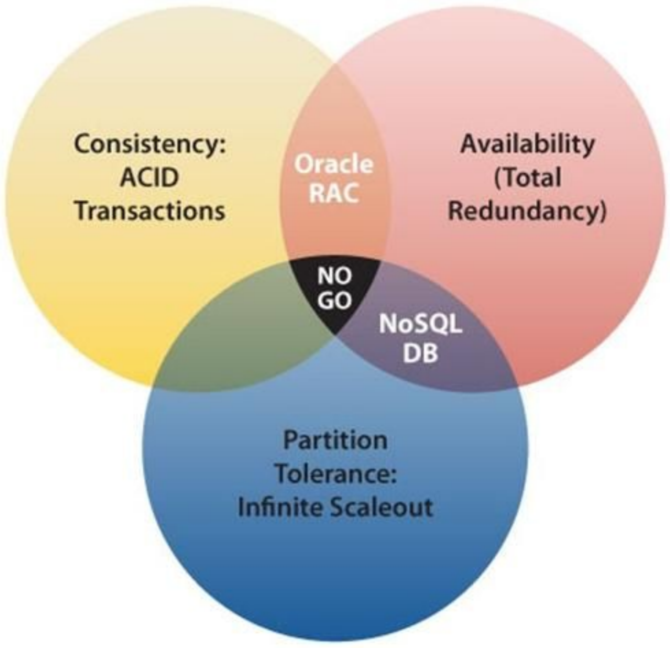

Consistency: ACID Transactions
Oracle RAC
Availability (Total Redundancy)
NO GO
NoSQL DB
Partition Tolerance: Infinite Scaleout

Figure 2.1: CAP Theorem Diagram [bigdatanerd (2011)]

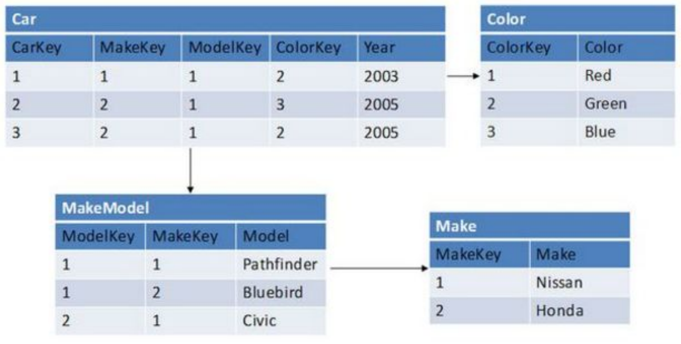
In a recently revised version in 2012 Brewer (2012), Brewer stated that this understanding is actually misleading. "*The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans for operation during a partition and for recovery afterward, thus helping designers think about CAP beyond its historically perceived limitations." - Eric Brewer*

2.2.2 Data schema

Data schema defines how the data is stored in the database. A schema usually contains information about primary keys, indexes, field types, relationships between data.

Relational Database

In SQL databases, data is essentially stored in groups of tables. Tables consist of columns and rows and represent either entities or relations between entities. Normalizing is usually utilized to prevents data duplicates and data inconsistency. Multiple tables can be accessed by "joining" them in a single query. It is impossible to add data before defining the table schema. Figure 2.2 shows a typical relational database model for a car database.



Car

CarKey	MakeKey	ModelKey	ColorKey	Year
1	1	1	2	2003
2	2	1	3	2005
3	2	1	2	2005

Color

ColorKey	Color
1	Red
2	Green
3	Blue

MakeModel

ModelKey	MakeKey	Model
1	1	Pathfinder
1	2	Bluebird
2	1	Civic

Make

MakeKey	Make
1	Nissan
2	Honda

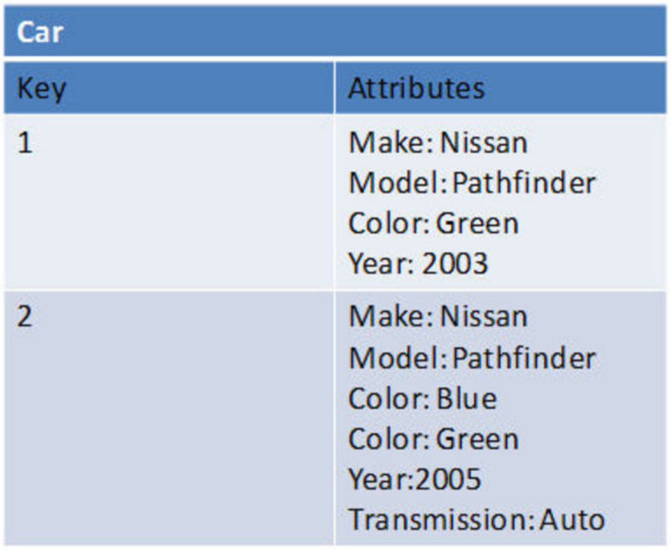
Figure 2.2: A typical relational database model [Bain (2009)]

Schemafree Database

While with traditional SQL databases the data schema always has to be predefined, NoSQL databases come with flexible schema design and basically belong to one of these four categories:

Key-value databases Key-value databases store data as pairs keys and values. Data can be unstructured and can only be queried by the keys. The most popular representative of this type is Redis. Amazon also uses its own key-value database - DynamodB - to store shopping carts. Key-value databases are most suitable for storing and looking up simple and large datasets at high speed. Figure 2.3 show the key-value database model of a car database.

Graph databases Although NoSQL databases excels in storing and sharding large amounts of data, they usually lack the ability to model relationships between datasets. One solution for this problem is graph databases.
Over the years, data has been connected through joins between tables in relational databases. However, there is a limit to how much joins can be utilized to show relationships in the real world. Often the queries can get too complicated making the lookups inefficient. Moreover, it is not possible to include attributes of relationships in joins. One of the notable representatives of graph databases is Neo4J. The main points here are nodes and edges. The nodes are entities which can assume many roles and participate in any number of relationships. The edges represent relationships between nodes and can have attributes. These attributes help specify the relationships. Graph databases has been proved to be extremely helpful, especially for



Car	
Key	Attributes
1	Make: Nissan Model: Pathfinder Color: Green Year: 2003
2	Make: Nissan Model: Pathfinder Color: Blue Color: Green Year:2005 Transmission: Auto

Figure 2.3: A key-value database model [Bain (2009)]

consumer and campaign surveys, fraud detection and social network applications. By using this kind of database, it is, for example, feasible to identify the groups of products usually purchased together by the same customers. Campaigners can also use feeds from social networks to determine the opinions of different target groups and hence will be able to implement correct strategies. Facebook, Google and Twitter all have their own built-in graph technologies, which have equipped their own social networks with many new possibilities. One of them is "Graph search". Human thinking is inherently visual. Graphs are, in fact, used everywhere, not only limited to scientific and academic fields, but also in daily lives to visualize data, which makes them easier for human beings to comprehend.Thinking in graphs is a natural way to connect data and expand possibilities, which can be useful in making decisions. Figure 2.4 shows a small social network graph. With the aid of "Graph Search" one can, in this case, determine all the sushi restaurants that one's friend likes.

Document databases Document-oriented databases store data as collections of documents in JSON files. Each record together with its associated data are treated and stored as a single document in the database. This reduces the needs for joins and complex operations to access the data. Documents are schema-free and contain pairs of key-value as attributes. The values can be in any formats: number, string, date or even document. This flexibility enables storing unstructured data, which is particularly useful nowadays, when collecting big datasets from many sources in various formats is a common thing. By storing documents in collections, data of a specific themes, e.g. student information, blog posts,.., can be store, search for and retrieved more efficiently.
The flexibility and robustness in querying data can also be achieved here. The data can be

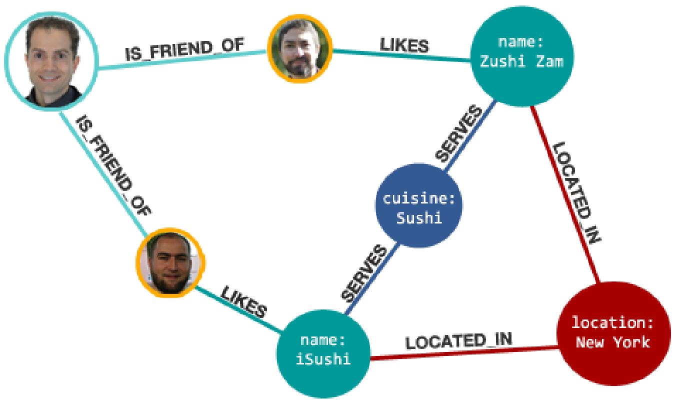
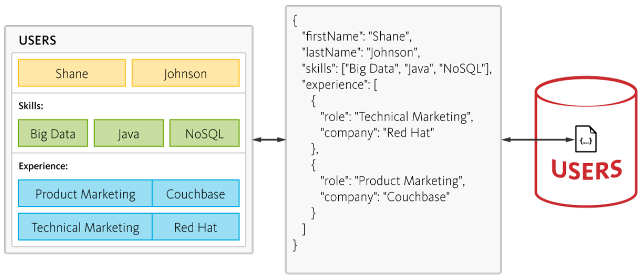


Figure 2.4: A graph database model [Eifrem und Rathle (2013)]

queried using both the keys and values like with relation databases. The parameters that can be passed to the queries are also various: from string, number, arrays of specific values to ranges of values.
Examples of NoSQL document databases include CouchDB, MongoDB, which is used within the context of this thesis for testing purposes and RavenDB. CouchDB and RavesDb both use JSON datas store documents while MongoDB uses BSON (Binary JSON) that enables binary serialization. Figure 2.5 shows an example of document database.

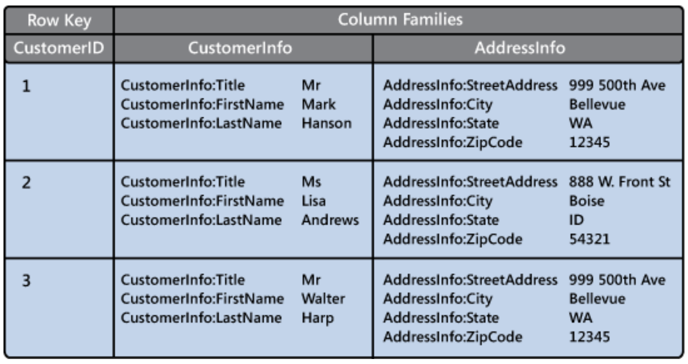


```
{
  "firstName": "Shane",
  "lastName": "Johnson",
  "skills": ["Big Data", "Java", "NoSQL"],
  "experience": [
    {
      "role": "Technical Marketing",
      "company": "Red Hat"
    },
    {
      "role": "Product Marketing",
      "company": "Couchbase"
    }
  ]
}
```

Figure 2.5: Example of a document database [Basel Kayyali und Kuiken (2013)]

Column-oriented databases Column-oriented databases use column families and rows to store data. At first sight, column databases appear very similar to relational databases. They both contains columns and rows in tables. The main difference lies at the column families. A column family is a group of related data which is often accessed together and can be described as a container of rows. Each row is identified by an unique key and has multiple columns. In column databases, rows don't have to contain the same columns. Only the column families must be defined upfront, the columns can be added later to any row and occupy zero bytes

if they don't have any information in themselves, which is a very efficient way to spare disk spaces. It is possible to have multiple versions of the same data due to the introduction of timestamps. The data with the most recent timestamp is considered the latest and will be read first in a query. A column family is how the database store data on the disk. All data in a single column family is stored together at the same location.
Column databases are restricted on how the data can be retrieved. Data can't be queried by column or value, only by row key. Joins are also not supported due to the fact that it requires an overview of the whole database, which is possibly divided to many nodes in a cluster, to locate the needed datasets.
Column family databases are meant for storing and processing large batches of data on a large number of machines, which is impossible for relational databases. Changes are first written into a commit log before being implemented. This helps determine the changes needed to apply after a system failure. Examples of this database type are Google's BigTable, Cassandra, Hbase. Figure 2.6 shows how the customer and address information can be stored in a column database.



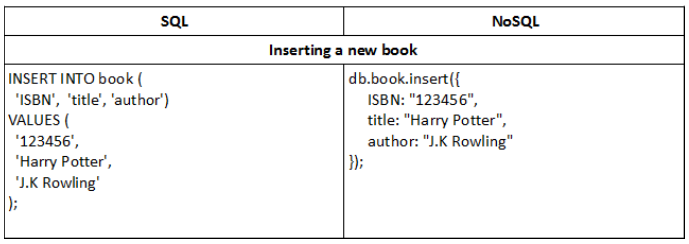
Row Key	Column Families	
CustomerID	CustomerInfo	AddressInfo
1	CustomerInfo:Title Mr CustomerInfo:FirstName Mark CustomerInfo:LastName Hanson	AddressInfo:StreetAddress 999 500th Ave AddressInfo:City Bellevue AddressInfo:State WA AddressInfo:ZipCode 12345
2	CustomerInfo:Title Ms CustomerInfo:FirstName Lisa CustomerInfo:LastName Andrews	AddressInfo:StreetAddress 888 W. Front St AddressInfo:City Boise AddressInfo:State ID AddressInfo:ZipCode 54321
3	CustomerInfo:Title Mr CustomerInfo:FirstName Walter CustomerInfo:LastName Harp	AddressInfo:StreetAddress 999 500th Ave AddressInfo:City Bellevue AddressInfo:State WA AddressInfo:ZipCode 12345

Figure 2.6: A column family database model [Microsoft]

2.2.3 Data query and data types

While traditional RDBMS SQL as a common language to query data, various NoSQL databases have different ways to query data.
Though slightly different in their syntaxes, all traditional databases use the same SQL queries, e.g. INSERT, UPDATE, DELETE, to modify data. On the contrary, how a NoSQL database queries the data relies heavily on what kind of database it is. Figure 2.7 shows how inserting a new book into a book database differs between SQL and NoSQL document database. The difference is comprehensibile. While the SQL database use common SQL to insert a new row of data, the NoSQL document database insert the new book record as a document which contains pairs of key-value as attributes.

Since NoSQL databases are schema less, there are no limits on what kind of data can be added to the database. Various types of data can be added directly to the databases without



SQL	NoSQL
Inserting a new book	
INSERT INTO book ('ISBN', 'title', 'author') VALUES ('123456', 'Harry Potter', 'J.K Rowling');	db.book.insert({ ISBN: "123456", title: "Harry Potter", author: "J.K Rowling" });

Figure 2.7: Inserting a book into a book database in SQL and NoSQL document databases.

having to comply with consistency rules. They can range from strings, numbers, documents to media files. The handling of different types of data is defined later in the application logic layer. This approach helps increase the performance of the database but requires more programming in the application logic.
With relation databases being very strict on data consistency, only the data types defined in the schema are allowed. This limit, on the other hand, helps reduce the need to handle data specifically in the application.

2.2.4 Scalability

Scalability has become more and more important through years. With the daily lives and business world now are powered by the Internet and new technologies - cloud, mobile, social media - most of the applications have turned Internet-based with millions of concurrent users. A system now has to be highly responsive, always available, is able to handle semi structured and unstructured data and can adopt new technologies quickly and easily. To meet these requirements scaling systems is unavoidable.
There are basically two strategies to scale a system: vertically and horizontally. There are also cases where these two strategies are combined.

Scale up A system can be scaled vertically by upgrading its hardwares, e.g. more powerful CPU, more RAMs, to boost the performance. This is a typical solution to scale up systems that use relational databases. The system will then be able to handle additional increased workloads. However there is a point where it is no longer feasible to improve the system through this way. It is either due to the limitations of the hardwares or the high expenses of such large servers.

Scale out One of the attractive features of NoSQL databases is their ability to be available all the time, which is crucial for web based applications that always has to deal with thousands of requests from clients. High availability and high performance are often achieved through scaling out the system, which simply means adding more servers to the network. This way, the system can decrease the workloads on each single node in the network. The data will then be divided into smaller units and spread among multiple servers.
Scaling out relation databases are not easy due to the complication of partitioning data. In

relational databases, joins are often required to retrieve data, which is not always possible when the tables needed are stored on different servers.

Data partitioning and sharding Data partitioning is required to spread data among multiple nodes. Using the right strategy to partition data can help enhance the availability and speed of the whole system and vice versa.
Vertical partitioning means dividing a table in multiple tables with fewer columns. Each of these new tables is stored on a single node. This is not always helpful, especially when the original table has billions of rows. Each of the resulting tables, in this case, can still be too large for a single node to handle.
Horizontal partitioning or *Sharding* divides a table in rows and store these rows on different nodes. This method reduces the burden on individual servers. Each server knows the entire table's structure, but only contains small sets of rows.

2.3 NewSQL

Despite their growing popularities over the past few decades, NoSQL databases still cannot totally replace traditional SQL databases.*"A realization that distributed databases have to make a choice between maintaining strict consistency (ensuring ACID Date und Darwen (1997) updates) and being highly available (tolerating outages) produced an impetus a few years ago to create a new breed of clustered databases."* Doshi u. a. (2013) While SQL databases provide reliability and data consistency through ACID transactions, NoSQL databases opt for high availability and speed. They settle for eventual consistency instead of constant consistency. Applications utilizing NoSQL databases often have to take on the responsibility of correctness and recovery in case of system or power failures.

The introduction of NewSQL databases to the market in recent years presents more choices for consumers. For organizations that have to deal with the explosion of daily data volumes, NewSQL pose an appealing alternative. While maintaining the relational structure, SQL as data query language together with ACID properties, this hybrid type of databases also delivers high performance and capacity through scalability.

2.3.1 Summary

The introduction of NoSQL databases doesn't mean the demise of traditional relational databases. Indeed, they should be considered more as an alternative of relational databases. Depends on the purposes of the applications, suitable type of databases can be chosen. When availability and speed are required, one can not go wrong with NoSQL databases. On the other hand, relational databases are probably the best choice when data consistency matters most. Other factors like the programming skills of the database designers, expenses, hardware capabilities should also be taken into consideration.

3 Analysis

This document, as introduced before, serves to present the main differences between the two databases MongoDB and VoltDB, spanning from comparing general features to real-time performance evaluation.

This chapter is dedicated to show an overview of both of the databases in an attempt to explain the reason behind the choices of MongoDB and VoltDB as test databases. It is then concluded by a tabular comparison of general features of the two databases.

3.1 MongoDB as NoSQL Representative

The first candidate for the evaluation tests is MongoDB. MongodB database belongs to the document-oriented databases category with its first version written in C++ and released in 2009. Aboutorabi u. a. (2015) Data is stored as documents in BSON files. Documents are not required to have the same schema. This feature enables partitioning data in several datasets and storing them on different servers for performance purposes. *"MongoDB database focuses on four characteristics of flexibility, strength, speed and ease of use."* Aboutorabi u. a. (2015). Besides, there are a number of drivers supporting different programming languages.

The reason why MongoDB is selected for the tests is because of its wide popularity. Many projects are based on this database including the Guardian news, Forbes business magazine, the New York Times, ebay, Global Financial Services Company, McAfee, Adobe,.. Different projects chose MongoDB for different reasons, depending on their needs. Some chose it due to the document schema being suitable for storing their contents. Other chose it for its scalability and support for distributed systems.

3.2 VoltDB as NewSQL Representative

One of the worth-mentioning representative of NewSQL databases and also a candidate for this evaluation is VoltDB. *"It's the latest database designed by Michael Stonebraker, the database pioneer best known for Ingres, PostgreSQL, Illustra, Streambase, and more recently, Vertica. But interestingly, in this go-around, Stonebraker declared that he has thrown "all previous database architecture out the window" and "started over with a complete rewrite".Stonebraker u. a. (2007)"* Bernstein (2014)

According to the official website, VoltDB claims to be an in-memory database, which increases the speed of processing data considerably, and to support SQL, ACID, stored procedures and interestingly HADOOP also. With these promising features, it would be interesting to see how VoltDB performs against a NoSQL representative like MongoDB.

3.3 MongoDB vs VoltDB

3.3.1 General Comparison

The table 3.1 shows a theoretical and general comparison between MongoDB and VoltDB. The comparison criteria are various, ranging from basic features such as database model, data schema, query languages, programming languages and APIs supported to extensional ones like partitioning and map-redude support.

Feature	**Description**	**MongoDB**	**VoltDB**
Database type	The type of database	NoSQL document database	NewSQL in-memory datbase
Implementation language	The language use to develop the database	C++	Java
Data schema	Is a data schema required?	no	yes
Database model	How a database stores data	Document-oriented	Relational
Query language	The language used to query datas	MongoDB database query language	SQL
Programming languages supported	Programming languages supported by the database	C C++ C# Java Javascript Perl PHP Python Ruby Scala Erlang (not officially supported)	C# C++ Erlang Go Java (packaged with VoltDB) Javascript PHP Python Ruby
APIs and drivers supported		Java API JDBC RESTful HTTP API	Java API JDBC RESTful HTTP/JSON API
Operating systems supported	Operating systems on which the database can be installed and run	Linux Mac OSX Windows	Linux Mac OSX CentOS Red Hat
Partitioning	Partitioning method	Sharding	Sharding
Replication	Replication method	Master-slave	Master-slave
Map-Reduce	Does the database support Map-Reduce?	yes	yes

Consistency	Consistency model of the database	BASE	ACID
Transaction	Does the database support transactions?	no	yes
Indexes	Does the database suport indexes?	yes	yes
Stored procedures	Does the database support stored procedures?	no	yes
Multi-users	Are multiple users possible?	yes	yes
Web interface	Does the databse possess a web interface?	yes	yes
Real time analytics	Are real time analytics supported?	yes	yes
Cloud platforms supported	The cloud platforms supported by the database	Amazon EC2 DigitalOcean dotCloud Joyent Cloud Modulus Rackspace Cloud Red Hat OpenShift VMWare Cloud Foundry Microsoft Azure IBM SoftLayer	Amazon EC2 IBM Softlayer Microsoft Azure HP Helion Google Compute Engine Open Stack Microsoft Server 2012 VMWare Vcloud IBM BlueMix Red Hat OpenShift Pivotal Cloud Foundry
Online backup	Is online backup possible?	yes	yes
Logging	Does the database possess logging functionatlity	yes	yes
Text search	Can the database perform text-search?	yes	no
Durability	Is the database able to preserve the commited data in case of system failure?	yes	yes

Table 3.1: MongoDB and VoltDB in comparison

4 Evaluation in real life application specific scenarios

4.1 Goals

The introduction of the NewSQL databases presents the possibility of consistent ACID transactions from traditional relational databases together with scale-out support, one of the most attractive features of NoSQL databases, which increases availability and performance of systems. This raises the question if NewSQL databases, in this case VoltDB, can eventually be considered an alternative for NoSQL databases. This thesis attempts to detect the differences in their performance in different test scenarios and uncover the best uses for each database.

The test scenarios presented in the context of this thesis are E-Commerce and Social Networks.

SQL databases are often the top choice for online stores due to ACID transactions, which ensures data consistency, and relations between tables, which enables joins and makes it possible to perform complex queries involving multiple tables (e.g. retrieving all products that customer A purchased this year together with their delivery status). Due to this fact, it is interesting to see how MongoDB-a NoSQL database, come up against a relational database like VoltDB in its own field.

In the case of Social Networks, it is crucial to have a fast to respond, highly available database system. It is usually horizontally scalable and supports map-reduce to aggregate through large amounts of data in order to deliver fast answers. In the past, traditional databases had difficulty in partitioning data to scale the system horizontally. This reduces the system performance tremendously and it is easy to overload the whole system. With the appearance of NewSQL databases, VoltDB in this situation, it now seems possible to have both transactions and sharding ability at the same time. Since NoSQL has dominated in Social Network applications for years, it is time to investigate if the new kind of SQL databases come up to the task and is finally able to compete with NoSQL databases, in this case MongoDB, in Social Network territory.

Because the two databases are very different in their structure nature, it is crucial to test them in their own best circumstances in order to evaluate them fairly correctly. That means for relational model, the tables should ensure third normalization. The structures of both databases should be as close as possible to ensure correct evaluation.

For both of the databases, Java API driver is used to connect, send queries and collect results. The reason of using the same kind of API driver is to avoid unnecessary overhead in processing the data which can slow down the test runs and make the results incorrect.

4.2 Equivalences in structure between MongoDB and VoltDB

In order to model the test scenarios in MongoDB and VoltDB precisely without giving any of them advantages or disadvantages over the other, a conversion between their data structures should be taken into consideration.

Table 4.1 [MongoDoc] shows their basic equivalences..

SQL	**MongoDB**
database	database
table	collection
row	BSON document
column	field
index	index
joins	embedded documents linking (e.g. ids)
primary key	primary key
primary key can be chosen from unique columns or column combinations	**_id** as default primary key
aggregation	aggregation pipeline

Table 4.1: SQL and MongoDB - basic equivalences

4.3 Test scenarios

4.3.1 E-Commerce

The test scenario is based on the concept of a real-life online store. The test make no attempts to cover all aspects of an online store, but only parts of it. The parts covered are: products, products categories, users and reviews of the products. The whole test is performed on a single node. The tests range from basic performance operations, such as inserting, retrieving and deleting data to other aspects which are important to an online store like transactions and joining multiple tables. Most of the tests are performed on a database storing 1000 products and 2000 reviews, except for the INSERT and DELETE performance tests.

Related works

In the past years, there are many related works which compares SQL and NoSQL databases in an ecommerce scenario, which usually include MongoDB, . Some of them are: Aboutorabi u. a. (2015), Parker u. a. (2013), Boicea u. a. (2012), Li und Manoharan (2013). Table 4.2 summarizes of these works.

Work	**Summary**
Performance evaluation of SQL and MongoDB databases for big e-commerce data - S. H. Aboutorabi and M. Rezapour and M. Moradi and N. Ghadiri	The paper Aboutorabi u. a. (2015) focuses on testing MongoDB against traditional SQL databases, which is SQL Server in this case, for the ecommerce purpose. The experiment uses a typical E-Commerce schema, containing entities like customer, product, product category, order, shipper and supplier, and includes performing basic operations, such as INSERT, UPDATE, SELECT and DELETE on both databases, followed by aggregate and non-aggregate function queries. According to the results, MongoDB performs better for the most part, except for some aggregate functions.
Comparing NoSQL MongoDB to an SQL DB - Parker, Zachary and Poe, Scott and Vrbsky, Susan V.	The authors of the paper Parker u. a. (2013) also come to the same conclusions after conducting a test experiment on MongoDB and SQL Server to observe their performance in the case of a modest-sized structured database. The results of MongoDB come out mostly equally or even better than its counterpart, excluding aggregate functions.
MongoDB vs Oracle – Database Comparison - A. Boicea and F. Radulescu and L. I. Agapin	In Boicea u. a. (2012), MongoDB is compared with Oracle SQl database on various fronts: theoretical features, syntaxes, restrictions, integrity model, distribution, query and insertion benchmarks. It is concluded that MongoDB is easy to use, flexible with storing different data structures, faster and supports map-reduce well while Oracle is slower albeit strictly consistent, has relations and joins, which enables complex queries.

A performance comparison of SQL and NoSQL databases - Y. Li and S. Manoharan	The paper Li und Manoharan (2013) aims at investigating the performance of various SQL and NoSQL databases in the key-value store aspect. The participants are: MongoDB, RavenDN, CouchDB, Cassandara, Hypertable, Couchbase and Microsoft SLQ Server Express. Fundamental operations:read, write, delete and instantiate are performed on key-value store implementations on the chosen databases and evaluated. Additional operations, which are also taken into consideration, are iterating through keys and iterating through values. The results show that not all NoSQL representatives perform better than their SQL counterpart. RavenDB and CouchDB are slow on read, write and delete operations. Cassandra doesn't perform well on read operations, is however good when it comes to delete and write operations. Of all the NoSQL databases presented, MongoDB and Couchbase are the fastest two overall. Couchbase doesn't, however, support iterating through keys and values. When iterations are not required in the applications, Couchbase comes out as top choice. Otherwise, MongoDB proves to be a well-rounded solution, coming second in performance after Couchbase.

Table 4.2: Related works

Test data

Since the test scenario aims at online stores, there's no better test sample than Amazon, one of the leading E-Commerce company. Founded in Seattle, Washington, United States in 1994 as an online bookstore, the internet-based retailer then diversified the products they sold and has since opened branches in other countries. In 2015, Amazon was evaluated as the most valuable retailer in the United States by its market capitalization, surpassing Walmart's.

The dataset used for this test is a snapshot of Amazon database provided by University Stanford J. McAuley 2015. It contains 142.8 million products reviews from May 1996 till July 2014. This dataset provides not only reviews but also metadata of products. The following product categories: books, movies and video games are used. Figures 4.3.1 and 4.3.1 shows how sample metadata and a sample review look.

Metadata sample

```
{
"asin": "0000031852",
"title": "Girls Ballet Tutu Zebra Hot Pink",
"price": 3.17,
"imUrl": "http://ecx.images-amazon.com/images/I/51fAmVkTbyL._SY300_.jpg",
"related": "also_bought": ["B00JHONN1S", "B002BZX8Z6"],
"also_viewed": ["B002BZX8Z6", "B00JHONN1S", ],
"bought_together": ["B002BZX8Z6"] ,
"salesRank": "Toys & Games": 211836,
"brand": "Coxlures",
"categories": [["Sports & Outdoors", "Other Sports", "Dance"]]
}
```

Review sample

```
{
"reviewerID": "A2SUAM1J3GNN3B",
"asin": "0000013714",
"reviewerName": "J. McDonald",
"helpful": [2, 3],
"reviewText": "I bought this for my husband who plays the piano. He is having
a wonderful time playing these old hymns. The music is at times hard to read
because we think the book was published for singing from more than playing
from. Great purchase though!",
"overall": 5.0,
"summary": "Heavenly Highway Hymns",
"unixReviewTime": 1252800000,
"reviewTime": "09 13, 2009"
}
```

4.3.2 Social Network

According to Wikipedia, *"A social network is a social structure made up of a set of social actors (such as individuals or organizations), sets of dyadic ties, and other social interactions between actors."* - Wikipedia
Social networks are not only a tool for maitaining personal relationships but also help with networking for business purposes. Some examples of popular social networks are Facebook (with more than one billion users), Twitter, LinkedIn (which offers opportunities in business and professional networking).
The reason this scenario is chosen for testing in the scope of this thesis is because of the popularity of social networks in daily lives globally. Besides, it also provides the opportunity to evaluate the test databases in a distributing environment, which is very different from a standalone system in the ecommerce scenario and thus require different tests tailored for this

environment. Social network differentiate from each other in scales and purposes. Small social networks operate nationally and therefore require less capacity. Large social networks, like Facebook and Twitter, usually operate on global scales, have an enormous amount of users and have multiple data centers in different locations to provide capacity, speed and high availability.

Related works

Related works, which are used as references for designing the test queries in the social network scenario, are listed in table 4.3.

Work	**Summary**
Benchmark: PostgreSQL, MongoDB, Neo4j, OrientDB and ArangoDB - ArangoDB	In ArangoDB, basic operations and social network related queries (e.g. get friends of friends, get shortest path) are tested on several SQL and NoSQL databases: MongoDB, ArangoDB, PostgreSQL, OrientDB and Neo4J (one of the leading graph databases), using data of user profiles and friendships from a snapshot of the social network Pokec provided by Standford University.
MySQL vs. MongoDB: The Pros and Cons When Building a Social Network - Gen	The article Gen gives a short comparision of MongoDB and MySQL in a social network environment. Even though MongoDB is faster at retrieving data, MySQL has the advantage of being able to handle relations, which are essential in social networks. The lack of supporting relations in MongoDB can cause data duplication and data inconsistency. Therefore, MySQL, according to the author, is a better choice.
Analysis of data management and query handling in social networks using NoSQL databases - A. B. Mathew and S. D. Madhu Kumar	Representatives of our types of NoSQL (document, key-value, column family, graph): Hadoop/HBase, Cassandra, MongoDB, CouchDB, DynamoDB, Riak, Voldemort, Neo4J, FlockDB, InfoGrid, AllegroGraph and OrientDB, which are used in well-known social networks, such as Facebook, LinkedIn, Twitter, MySpace, Foursquare, Flickr and Friendfeed, are tested in the paper Mathew und Kumar (2015) in many aspects, ranging from scalability, concurrency control, consistency in storage, availability during partitioning, durability, implementation language to transactions. Features like fast query processing and data storage are given primary importance. The test results in Neo4J performs better than other NoSQL databases during insert and read operations. It is also shown that graph databases play an important part in social networks.
Benchmarking Graph Databases - ISTC	In the benchmark in the articlevISTC, VoltDB competes again Neo4J, MySQL and Vertica in graph management and analytics. Two queries, PageRank and Shortest Paths, are tested on each system. It is shown in the results that relational databases outperform or match Neo4J in most cases.

Table 4.3: Related works

Test data

The data used for this test scenario comes from Standford University and is a snapshot of Facebook data Standford. This dataset consists of circles of friends (friend lists) on Facebook. It also provides profiles of users in the circles.

4.4 Evaluation criteria

Before conducting the comparison tests of MongoDB and VoltDB, it is necessary to define the criteria, upon which each database is evaluated. These general criteria are also needed for designing the suitable tests for each scenario. Because of the different nature and requirements in the two chosen test scenarios, for each comparison criterion, the tests are designed differently for each scenario.

4.4.1 Data model

This comparison show how MongoDB and VoltDB differentiate from each other in their data modeling while interpreting the same test scenario.

4.4.2 Performance

The performance tests cover the CRUD operations, namely CREATE, READ, UPDATE and DELETE, in each database in the same scenario. Queries that are related to the test scenario are also tested and measured.

4.4.3 Functionality

The differences in functionality between MongoDB and VoltDB are shown here. Depending on the test scenario, suitable functionalities are chosen to be tested. They can be decisive factors to determine which database is best suited for which scenario.

4.4.4 Transaction

ACID transactions are one prominent feature in SQL databases. Due to the fact that this thesis compares a NoSQL and a SQL database, it can be interesting to see the performance differences when the same query is performed both as transaction and non-transaction.

4.4.5 Security

Security is an important matter when multi-users are allowed. In both scenarios, multiple users are present at any given time. Maintaining controls and restrictions on users are therefore significant. Only users with given roles are authenticated to access certain data and resources. Failing this can lead to leaks of crucial information. It is hence imperative to evaluate each database's ability to secure the data in different scenarios.

4.4.6 Distribution

One of the attractive points of NoSQL databases is the ability to scale horizontally across a cluster of nodes. The data can be partitioned and distributed to multiple nodes, which reduces the burdens on a single node and also raises performance of the whole system. MongoDB and VoltDB both support partitioning and therefore can be tested in a cluster environment. Features to be tested include replication and single-partitioned procedure.

4.5 Evaluation tests

4.5.1 E-Commerce

The test scenario is based on the idea of a real-life online store. The test make no attempts to cover all aspects of an online store, but only parts of it. The parts covered are: products, products categories, users and reviews of the products. Tests are performed on a single node.
Test system
The testing virtual machine has the following system configuration:

- Processors: 4 Cores
- Memory: 16GB
- Hard disk: 80GB
- Operating system: Ubuntu 14.04

Testing databases are:

- MongoDB 3.2
- VoltDB 6.5

Data model

VoltDB
Due to the fact that VoltDB is a SQL database, a schema has to be created before data can be inserted into the database. Figure 4.1 shows a data model for VoltDB in the ecommerce test scenario. There are four tables in total: product, review, user and category. The table "product"

is comprised of basic information about a product like productId, title, categoryId, price and imageURL. The table "user" contains only userId and userName and serves only as a lookup source for the table "review". The table "category" is linked to the table "product" by the foreign key "categoryId".

MongoDB

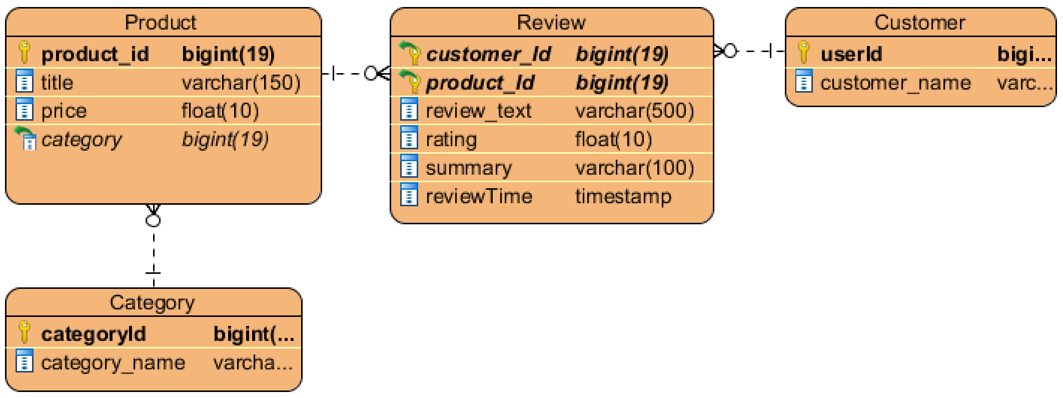


Figure 4.1: VoltDB schema

Although a schema is not needed for MongoDB, it is still necessary to decide how the data should be stored in the database. Ideally the structure of the database should come as close as possible to the structure of VoltDB's database for the comparison. A collection is created for each of the table in figure 4.1 respectively.

Performance

In the scope of this performance test, CRUD operations are performed includes: INSERT, SELECT, UPDATE, DELETE. These operations are performed only on the table *product*. Each of these operations are performed three times. INSERT and DELETE operations are performed on three different scales of data: 10, 100 and 1000 data records. Both SELECT and UPDATE are performed on 1000 product scale. The average time after three test runs is used as the final result. Table 4.4 shows the queries used for this test.

Testcase	**Query**	**MongoDB**	**VoltDB**
INSERT	Insert 10, 100, 1000 products	Single- write Bulkwrite	AdHoc SQL Stored procedure
SELECT	Return the products, whose prices are higher than \$10.00 and smaller than \$30.00 (performed on 1000 products).	Aggregation framework Map-Reduce	AdHoc SQL Stored procedure

UPDATE	Reduce the prices of products, whose prices are higher than \$10.00, by \$5.00 (performed on 1000 products).	Aggregation framework Map-Reduce	AdHoc SQL Stored procedure
DELETE	Delete 10, 100, 1000 products	Single-delete Bulkdelete	AdHoc SQL Stored procedure

Table 4.4: Performance test queries

Functionality

Joins over mutiple tables
Joining tables are an important factor in relation databases. By linking tables by their ids, it is made possible to retrieve and return related information from multiple tables.

Due to the lack of a relational data model, it is usually not easy for NoSQL databases to perform complex queries like SQL databases. Depending on the database, there can be workarounds to overcome this problem. In case of MongoDB, the solution is either embedded documents or references. Using embedded documents means the extra information is stored as a sub document in the main document. There is possibly redundant and duplicate data. Using references is more similar to the relations between tables in relational databases. An id of a document in a foreign collection is used as referencing id. If additional information is required from this foreign document, it is looked up by its id. This handling of references, however, is not conducted by the database as in the case of SQL databases. The retrieval of referencing ids and the lookup of respective documents needs to be implemented explicitly in the application programming. This can increase the complexity of the application and has bad impact on performance by making multiple round-trips to the database to retrieve all the required information. Since version 3.2 of MongoDB MongoDBBlog, left join can be included in an aggregate framework by using the *\$lookup* operator, which simplifies the lookups for references much and reduces the needs of handling references in application code.

Test case	**Query**	**MongoDB**	**VoltDB**
JOIN WITHOUT AGGREGATION	Return all product reviews from a specific user together with products' names given the name of the user.	Application handling Aggregation framework	AdHoc SQL Stored procedure
	Return all product reviews of a specific product together with the customers' names given the name of the product.	Application handling Aggregation framework	AdHoc SQL Stored procedure

Table 4.5: Join without aggregation queries

Table 4.5 shows the test queries. In MongoDB, each query is performed and measured twice. In the first run, the joins are handled directly in the application code. In the second run, the joins are performed by using the operator $lookup in an aggregate pipeline. In VoltDB, the queries are also performed twice, the first time as AdHoc SQL, and the second time as stored procedures.

Aggregation

Aggregation operations process data by grouping and performing a variety of operations on them to return a single value as result. Aggregation is often combined with joins.

Aggregation possibilities in MongoDB are: aggregate pipeline, map-reduce and single purpose aggregate functions. The idea behind aggregate pipeline is that of a data processing pipeline. Documents undergo multi-stage transformation and result in a single return value at the end of the pipeline. The stages are native operations and can be added if desired. The aggregate pipeline can also be used on sharded data. Starting from version 3.2, there is a new possibility of performing left equi-join in MongoDB by using the operator *$lookup* as a stage in an aggregate pipeline. This approach enables performing joins in the database and combining it with other processing in a single query. It results in less complex codes and less round-trips to the database, which helps boosting overall performance. Further information is available at Morgan (2015). Both cases: joins in application code and joins in aggregate pipeline are carried out.

Aggregation in VoltDB are usual SQL aggregate operations,such as SUM, GROUP BY, LIMIT, AVG. The queries are passed as AdHoc SQL and stored procedures.

Test case	**Query**	**MongoDB**	**VoltDB**
JOIN WITH AGGREGATION	Return the average rating and the number of reviews of a specific product, given the product name.	Application handling Aggregation framework	AdHoc SQL Stored procedure
	Return top 20 books with the most reviews.	Application handling Aggregation framework	AdHoc SQL Stored procedure
	Count the number of products in the category *Book* whose prices are between $25.00 and $50.00.	Application handling Aggregation framework	AdHoc SQL Stored procedure

Table 4.6: Join with aggregation queries

Just as in the case of joins without aggregate functions, the queries are performed in MongoDB both by handling joins in the application as well as inserting the $lookup stage into an aggregate pipeline. In VoltDB, the queries are carried out as AdHoc SQL and as stored procedures. Table 4.6 shows the test queries.

Indexes

Using indexes is to trade space for speed. By using indexes, a small portion of the dataset is stored in an easy to traverse form. Index stores ordered values of a specific field or set of fields. This ordering of values makes range-based search and equality match easier and more efficiently. Indexes are either unique or non-unique. In MongoDB, indexes are applied on collection level and can be used for any field or sub-field of a document. An index for the field _id is automatically created. In VoltDB indexes are specified in constraints and are created on columns of a table or expressions based on the table.

Test case	**Query**	**MongoDB**	**VoltDB**
Non-unique single index	Create a non-unique index on the field *price* in table *product*. Perform this query: Return 50 products priced between $25 and $50.	*find* query	AdHoc SQL
Unique compound index	Create a unique index based on the fields *product_id*, *customer_id* and *review_time* in table *review*. Get the *product_id* of a random product. Perform this query: Return all reviews of a specific product, sorted by the review time in descending order, given the *product_id*.	*find* query	AdHoc SQL

Table 4.7: Index test queries

An ecommerce database contains thousands to millions data records. In able to perform search queries efficiently, indexes should be applied to limit the number of search results and shorten the search time. Hence it makes sense to test indexes in ecommerce scenario. The tests measure the perfomance of the databases when using indexes and are performed as described in table 4.7. Normal *find* query is used to retrieve data in MongoDB since no joins are required and it is also more accurate to evaluate the effect of indexes on search queries without the performance boost of aggregation framework.

Text search

In a real life ecommerce scenario, it is quite common that customers search for products based on their names. Because of that, it is helpful to have effective and efficient text search in the database.

Dedicated text search is a functionality featured in MongoDB. With text search, finding data records based on a string field can be faster. In order to perform text search queries, text indexes are needed. Each collection should only contain one text index. This index can, however, involves a single field or multiple fields. Operator $text is then used to perform text searches on collections with text indexes. Text search can search for records containing any terms in a given list, look for exact matches and exclude records containing specific terms from the results. Text search can't, however, find records based on wildcard terms. There are workarounds for this problem though.

On the contrary, there is no specialized text search in VoltDB. Searching for string terms can still be performed using SQL terms, such as LIKE for wildcard matches, IN for items in a list, equal sign for exact matches.

Test case	**Query**	**MongoDB**	**VoltDB**
Text search	Search for records containing the term "java". *(Exact Match)*	*find* query	AdHoc SQL
	Search for records based on the wildcard "ava". *(Wildcard)*	*find* query	AdHoc SQL
	Search for records containing any of the following terms: "java", "coffee", "house" *Terms in a list*	*find* query	AdHoc SQL
	Search for records containing the term "java", except for records also containing the term "coffee". *Excluding term*	*find* query	AdHoc SQL

Table 4.8: Text search test queries

Table 4.8 shows the queries designed for text search test and the methods used in each database. Normal *find* query is used to retrieve data in MongoDB since no joins are required. The decisive factors here are the time measurements and the resulting sets of records.

Transaction

Transactions, which have ACID properties, ensure data consistency but can sometimes take longer to process than performing each operation separately and sequentially. To investigate the possible time differences in performance between transactional and non-transactional queries, a read-only transaction can be performed as described in table 4.9. The reason why a read-only transaction is chosen for testing is due to the fact that in real life online stores, reading from database are performed more frequently than writing.

MongoDB doesn't support ACID transactions and therefore performs each of the operations separately. Despite the lack of ACID transaction support, the write operation in MongoDB is atomic, but only on the level of a single document. When a single write operation modifies multiple documents, even if one document is already modified, it is likely that another process may interfere with the rest of the modifications. One way to prevent this is to use operator *$isolated* to keep other processes from interleaving once the first document is modified. This way the user only see the changes once all modifications are made. The above mentioned query is tested in MongoDB using aggregate framework due to its complexity, which involves many operations, such as: grouping document, joining collections, sorting and limiting the results. It is also necessary to evaluate the built-in framework in MongoDB, which is capable of joins, against traditional SQL transactions in term of performance and the results delivered. Since the test involves multiple joins, aggregation framework is used.

In VoltDB, it is possible to perform either each operation separately or the whole transaction as a stored procedure. Both cases are tested here to determine the time difference between transactions and non-transactions.

Test case	**Query**	**MongoDB**	**VoltDB**
Transaction	Deliver 20 products in *book* category which have an average rating of 5.0 and are sorted by price in ascending order	Aggregation framework	Stored procedure

Table 4.9: Transaction test query

Security

Multi-user and data consistency
In a real-life online shop, many users can log in and access the database at the same time. Therefore, a multi-user test is mandatory to determine if the data stay consistent when there are many users access the database at the same time. The test case is conducted as described in table 4.11.

Role-based access control
In general, roles are created and assigned to groups of users to grant privileges. By assigning roles to an user, an user can has access to resources (e.g. databases, collections,clusters) or permissions to perform specific actions (e.g. stored procedures). By default, VoltDB and MongoDB grants clients access to all databases without checking authentication. This feature needs to be switched on manually. Except for admin users, normal users should only be granted limited

access so that roles and users are kept under control and unwanted modifications of data can be avoided.

Other than user-defined roles, there are also roles predefined by the database. *"MongoDB provides built-in roles that provide the different levels of access commonly needed in a database system. Built-in database user roles and database administration roles roles exist in each database."* MongoDoc The admin database also contains additional roles: Cluster Administration Roles, Backup and Restoration Roles, All-Database Roles. Other built-in roles are: Superuser roles, which grant any user full privileges on all resources, and internal role. In VoltDB, when security is enabled, there are two roles predefined: *user* and *admin*. *"Administrator has ADMIN permissions: access to all functions including interactive SQL queries, DDL, system procedures, and user-defined procedures. User has SQL and ALLPROC pemissions: access to ad hoc SQL and all default and user-defined stored procedures."* VoltDBDoc The test for the built-in roles is described in table 4.11.

If the built-in roles don't satisfy the needs, customized roles are also possible. Both MongoDB and VoltDB support user-defined roles. According to MongoDoc, in MongoDB, a new role can be defined by having their privileges explicitly listed or by inheriting from other roles. Only a role created in *admin* database can be granted privileges on *admin* database, other databases, cluster resource and is able to inherit from roles in other databases and also from those *admin* database. A role created outside *admin* database only has access to its database. VoltDB offers the CREATE ROLE statement to create and grant a new role access to procedures and functions. The created roles can be specified in CREATE PROCEDURE statements to determine which roles are allowed to call which procedure. The WITH clause in a CREATE ROLE statement specifies permissions. Generic permissions are denied by default and need to be defined explicitly using CREATE ROLE statements. Permissions are accumulative, which mean when an user is assigned many roles, he has all permissions defined in these roles. Generic permissions include:

Permission	**Description**
DEFAULTPROCREAD	Read-only default procedures (SELECT)
DEFAULTPROC	All default procedures(SELECT, INSERT, UPDATE, DELETE, UPSERT)
SQLREAD	Read-only AdHoc SQL
SQL	All Ad-hoc SQL and default procedures
ALLPROC	All user-defined procedures
ADMIN	Full access to all system procedures, user-defined and default procedures, DDL statement and AdHoc SQL

Table 4.10: Permissions for user-defined roles in VoltDB [VoltDBDoc]

The test on user-defined roles is described in table 4.11.

One feature, which is only possible in VoltDB, is granting roles access rights to transactions, also known as stored procedures in VoltDB. If the transaction test proves that transactions are faster than non-transactional queries, VoltDB's ability to grant permission to stored procedures should be taken into consideration while determining the most suitable database for ecommerce.

Test case	**Query**	**MongoDB**	**VoltDB**
Multi-user	User A logs in and looks for a product P, whose price is higher than $25.00, its number of reviews and its average rating. Admin User logs in and apply a $5.00 reduction to products whose prices are higher than $25.00. User B logs in to write a product review for the product P. User A refreshes the view (sending the same request to the database again). The information of the product P, including its price and its number of reviews should already be changed.	*update* and *insert* queries combined with aggregation framework	AdHoc SQL

Role based access control	Enable access control. Create an admin user with full privileges on all resources. Log in as admin user, create a new user and grant this user read-only permission on all databases. Log in as the newly created user. Attempt to query for 25 products in the table *product*. The query should be performed successfully. Attempt to create a new review in the table *review*. The action should be denied due to lack of permission. Log in again as admin. Grant the above mentioned user read-write permission on table *review*. Log in as the normal user. Try creating a new review in the table *review*. The action should succeed.	*find* query	AdHoc SQL

	Enable access control. Create an admin user with full privileges on all resources. Log in as admin user, create a new role permitted to perform the following actions: SELECT on the table *product* and INSERT, SELECT on the table *customer*. Log in as the newly created user. Query for a random product. Attempt to change the *price* of that product. The query should fail. Make another attempt to create a new customer in the table *customer*. The action should succeed. Try deleting this newly created customer, permission should be denied.	*update*, *find* and *insert* queries	AdHoc SQL

Table 4.11: Security test cases

4.5.2 Social Network

The Social Network test scenario aims at evaluating the databases' ability to handle common situations and interactions in a typical social network environment.

Test system
The tests are conducted in a cluster, consisting of different virtual machines. A cluster consists of one config server and two shards for MongoDB and a cluster with two shards for VoltDB are used for most test cases instead of replication test. For MongoDB, three virtual machines are used. For VoltDB only two are used since a config server is not required. The specific test systems for replication test are discussed later.

All the virtual machines have the same system configurations:

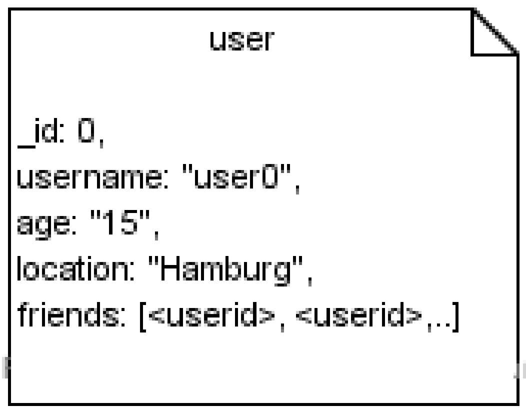
- Processors: 4 Cores
- Memory: 16GB
- Hard disk: 80GB
- Operating system: Ubuntu 14.04

Testing databases are:

- MongoDB 3.2
- VoltDB 6.5

Data model

Social networks are often depicted as large graphs with users as nodes and connections between users as edges. There are graph databases such as Neo4J which specilizes in storing data in graphs and performing graph-related queries like find friends of friends by traversing along edges or find shortest path between two given nodes. Since both MongoDB and VoltDB are not graph databases, data model is thus very important. An intelligent model of a social network makes conducting graph-related queries more effectively and efficiently and vice versa. In MongoDB, an user profiles are stored as documents containng the following information: user id, user name, age, location, highschool(optional since not all users have this feature) and friends. Instead of having their friendships stored in a different collection, users have the user ids of their friends stored in a *friends* array in their profiles as references to look up the profiles of these friends. As MongoDB is not a relational database and doesn't perform joining tables well, it is more efficient to store the friendships this way. Besides, it also helps lower storage cost since no separate collection for friendships are needed. There is only one collection *user* in the database. Figure 4.2 shows an example of how an user profile is stored in MongoDB.


user

```
_id: 0,
username: "user0",
age: "15",
location: "Hamburg",
friends: [<userid>, <userid>,..]
```

Figure 4.2: A document in MongoDB

For VoltDB, there are three tables: *user*, *friendship* and *post*. Table *user* stores user profiles while table *friendship* and *post* contain the friendships among users and user posts respectively. Figure 4.3 shows how the tables are modeled in VoltDB.

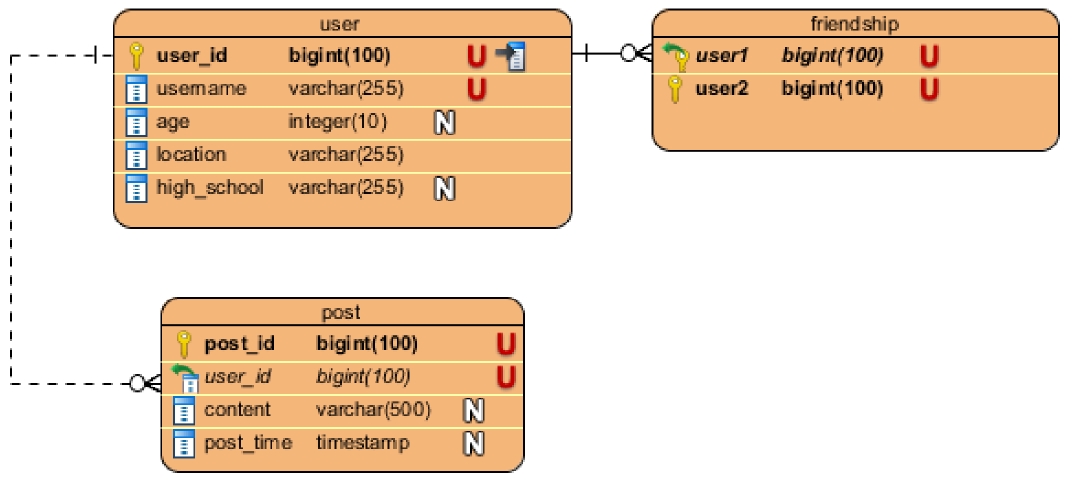


Figure 4.3: VoltDB schema

Performance

Social networks center around connections among people. The connections can be friendship, business. Either way, the focus on relationships among people leads to certain queries performed more often on a daily basis.

While the ecommerce tests are conducted on a single system, the social network tests evaluate performance of a database in a cluster environment. Partitioning method of test database is thus important. Depending how data is partitioned in a database system, the performance of processing queries can be affected. For example, if related information is on the same node, the retrieval of data is faster and vice versa. In this test scenario, all the performance tests focus on the two test databases' ability to scale horizontally and process sharded data in a cluster.

Sharding in MongoDB
Sharding is supported in MongoDB through a *sharded cluster*, which is composed of:

- *Shards*: data storing units.
- *Query routers*: also known as mongos instance, are media between client applications and shards, whose jobs are directing requests from clients to appropriate shards and return the results to client applications. One cluster can have more than one query routers.
- *Config servers* contain metadata of the clusters, which are used by query routers to find target shards for operations.

Figure 4.4 shows an overview of a sharded cluster in MongoDB.
Sharding in MongoDB is on collection level. Data of a collection is partitioned by a *shard key*. Shard keys are indexed fields existing in every document in the collection. Shard key values are divided into groups, also known as *chunks* and distributed evenly across the shards.

There are two partitioning methods for dividing shard key values into chunks: *range based partitioning* and *hash based partitioning*. Range based partitioning divides shard key values into ranges. Two shard keys with close values are likely to end up on the same shards. Defining ranges of shard key values can be done by using *tag aware sharding*. The advantage of this method is the possibility of data localization, which boosts speed and performance. The main disadvantage is the unevenly divided ranges, resulting in uneven distribution of workloads in the cluster. *Hash based partitioning* is more of a random method. The hash values of the shard keys are computed and used to create chunks. Two shard keys with close values are unlikely to be on the same shards. This approach ensures an even distribution of data across shards. However, because the distribution is random, a range query on shard key may result in having to query every shard to retrieve all the information. More details can be found on MongoDBDoc (2016).

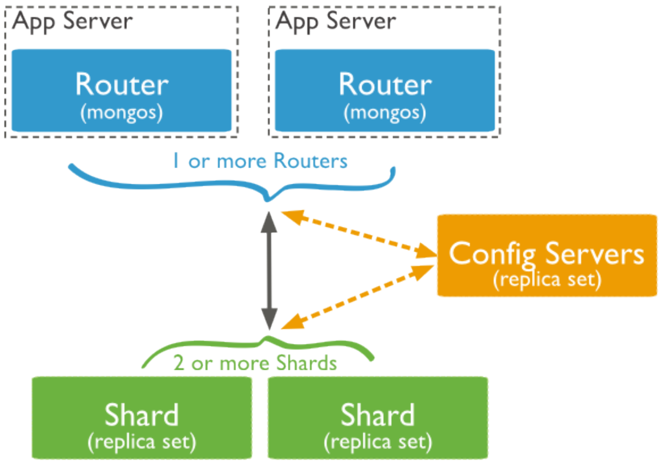


Figure 4.4: A sharded cluster [MongoDBDoc (2016)]

Sharding in VoltDB
In VoltDB, the partitioning is handled automatically based on a partitioning column chosen by the user. The values of the partitioning column are then hashed and associated to partitions. It is possible to have all partitions of a table on the same node as well as on different nodes. VoltDB partitions not only the data but also the processing of that data. This enables parallelism and provides improvements in performance. The partitioning can be specified in the schema file and loaded together with table definitions before inserting data into the database.

Testing
All the performances tests are performed for each partitioning method. In MongoDB, they are, as mentioned above, *range based partitioning* and *hash partitioning*. In VoltDB, it is partitioning a table based on a *partitioning column*.

Not all possible operations are tested here, only the main sensible ones that fits into a social network scenario. Table 4.12 shows the queries of each test case and the methods used in each database to perform them. Although *finding the shortest path* from one node to another node is often performed in graphs, it does not only just depend on the database performance but also on the algorithms used. Moreover, *finding the shortest path* is not a supported feature in MongoDB and VoltDB and therefore is not tested here.

Test case	**Query**	**MongoDB**	**VoltDB**
Single read	Return 1365 user profiles	*find* query	AdHoc SELECT
Single write	Insert 1365 user profiles	Bulkwrite	AdHoc INSERT
Count members of an age group	Return the number of users whose age between 25 and 35	*count* query	AdHoc SELECT
Friends of friends	Return friends of friends (only IDs). This test is performed on 333 users due to performance limitations of the test system	Aggregate Framework	AdHoc SELECT
Friends of friends with profiles	Return friends of friends (profiles included). This test is performed on 333 users due to performance limitations of the test system	Aggregate Framework	AdHoc SELECT
Mutual friends	Return user profiles of mutual friends of two given users	Aggregate Framework	AdHoc SELECT
Get friends in the same location	Return all friends of an user living in the same location	Aggregate Framework	AdHoc SELECT

Table 4.12: Social network performance test queries

The decisive factor which is taken into consideration is the time needed in milliseconds to fulfill each of the above tasks and the complexity of designing each query in each database.

Functionality

Aggregation
As data in a social network is sharded and stored on nodes of a cluster, aggregation methods are usually used to query data separately on different nodes. The result sets from all nodes are then grouped together to form a final result.

In MongoDB, there are two ways of aggregation in a distributing environment: *aggregation pipeline* and *Map-Reduce*. An aggregation pipeline operates on sharded collections and is a form of data processing pipelines. There are multiple stages in a pipeline. Documents are inserted into the pipeline as input and go through several filters and transformations, which result in an aggregated result. *Map-Reduce* is the usual way databases use to process and aggregate data in a distributing system. MongoDB also provides this method. Basically, *Map-Reduce* includes two main parts: a *map* phase to process the data and a *reduce* phase to combine the results of the *map* phase. Like other aggregation methods, *Map-Reduce* can filter input documents as well as sort and limit the results. Additionally, there is an optional *finalize* stage to alter the result the last time before showing the final result. *Map-Reduce* operates on sharded collections and can produce result as a sharded collection.

VoltDB also supports *Map-Reduce* to process data in a cluster. In fact, it is made possible by integrating VoltDB with *Hadoop*. *Hadoop* is an open source framework which can be used to manage and manipulate Big Data. Not only is *Map-Reduce* already included, but there is also a built-in distributed file system (HDFS) in *Hadoop* to handle distributing data. *Hadoop* can handle terabytes, even petabytes of data. *"This is possible because Hadoop separates the physical storage of data from the application interface for reading and writing."* The export of data to *Hadoop* in VoltDB can be automated. Configuring this is simple and doesn't require programming. Users specify tables in the schema as sources for export. Any data written to the specified tables at runtime is automatically transported to the VoltDB export connector, which sends the updates to the Hadoop destination. One advantage of this approach is that the whole process is automated, the users don't need to know the details of the destinations, to which the data is exported. Unfortunately, this method is not tested in the scope of this thesis due to the fact that the export functionality to Hadoop is not available in the community edition of VoltDB. Therefore, the queries are performed using normal SQL.

Testcase	**Query**	**MongoDB**	**VoltDB**
Aggregation	Return all users who used to study or is studying at the same high school together with their number of friends given the name of the school .	Aggregation framework Map-Reduce	AdHoc SQL

Table 4.13: Aggregation test

The query described above in table 4.13 is used to evaluate the end result and the time difference between aggregation methods in MongoDB and VoltDB.

Indexes and text search

Indexes help boost the speed of data processing. Indexing an attribute reduces the needed time to traverse through the values of that attribute and therefore makes the searches faster. It is very common to search for users or organizations on social networks based on their names. Therefore the text search functionality can be quite useful to get exact or partly correct matches.

Because usual types of indexes and text search were tested in the ecommerce scenario and there is not much difference between the two scenarios in applying these features, another type of indexes which may be useful in social networks is discussed and tested here is *partial index*. *Partial* index features in MongoDB and indexes only documents which fulfill specified requirements. Since not all the users in a social networks have the features listed in their profiles, it makes sense not to use some indexes on all the documents but only on a selected set. This type of index helps lower storage and performance cost since not all documents in the database are indexed.

Table 4.14 shows the query designed for the *partial index* test.

Testcase	**Query**	**MongoDB**	**VoltDB**
Partial index	Create a partial index on users with the feature *high_school*. Return all the users studying at a specific school given the school name.	*find* query	AdHoc SELECT

Table 4.14: Partial index test

Transaction

While most transactions in ecommerce database systems are read-only transactions, for social networks, write operations are almost as important as read operations. An example is Facebook. Everyday, there are millions of posts created and read by the users. This requires databases that can perform both read and write operations equally well in a cluster containing multiple nodes. Therefore, it makes sense to compare performance of non-transactional read-write queries and that of transactional read-write queries in cluster environment. Table 4.15 shows the content of the read-write transaction test and the methods used in each database.

Testcase	**Query**	**MongoDB**	**VoltDB**
Transaction	As an user write a post and publish it (which means the post is saved to the database). After publishing the above post, the page is refreshed and all the posts together with the new post are shown.	*find* query	AdHoc SELECT Stored Procedure

Table 4.15: Transaction test

Each database has to perform two operations for this test: *writing the new post to the database* and *returning all posts*. The test is performed in MongoDB as non-transactional separate queries and in VoltDB as non-transactional separate queries as well as one single transactional query (stored procedures).

Distribution

Replication
Replication provides high availability and redundancy. By having the same contents on more than one node, the whole system becomes more tolerant of individual node failures due to power outages as well as natural disasters and also offers data localization by having multiple copies of the same contents in different locations, which increases the processing speed when a local client connects to the local database. Since social networks handle millions of requests and big datasets every day, replica are needed to share the workload, secure the data and help provide high availability for the network. Because of this reason, replication is choosen as one of the factor to be tested in the social network scenario.

In MongoDB, replication exists in form of *replica sets*, whose example can be seen in figure 4.5. A *replica set* contains a *primary node* and several *secondary* nodes. The *primary* node receive all read/write operations and records all data changes in an *operational log*. The *secondary* nodes read this *operational log* and applies the changes to their data to match the *primary node* 's state. If the *primary* node is unavailable, a *secondary* node can elect itself to become the new *primary node*. An *arbiter* can also be added to the replica set. An *arbiter* holds no data and its only function is to maintain the quorum in the votings. It always stays an *arbiter* while a *primary* node can becomes *secondary* node and a *secondary* node can turn into a *primary* node through election. There are two forms of replications supported by VoltDB: *one way (passive)*

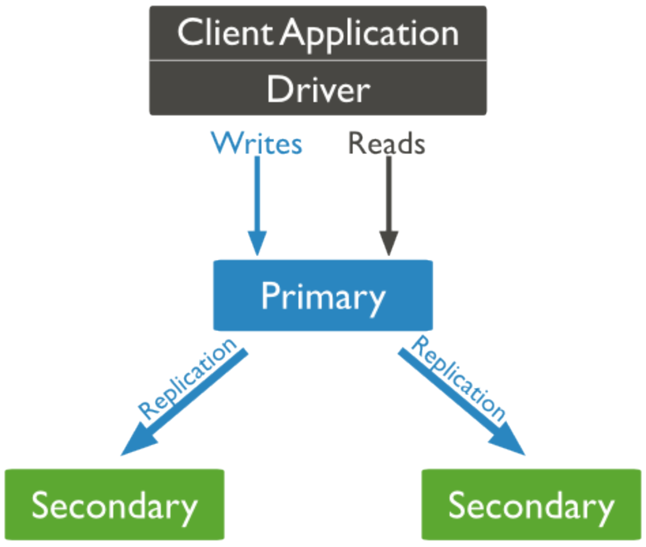


Figure 4.5: A replica set [MongoDoc]

and *two-way (cross datacenter)*. In *passive database replication*, the replication occurs only one way, from the *master* database to the *replica* database. To keep the data consistent between the *master* and the *replica* databases, the *replica* database is read-only, all the modifications have to be performed on the *master* database. The figure 4.6 shows how this type of replication works. On the contrary, in *cross datacenter replication*, the copying of data happens in both ways. There is no *master* database here, all databases with the same content are treated equally. Users can perform read/write operations on either database. The changes are then applied to the others.

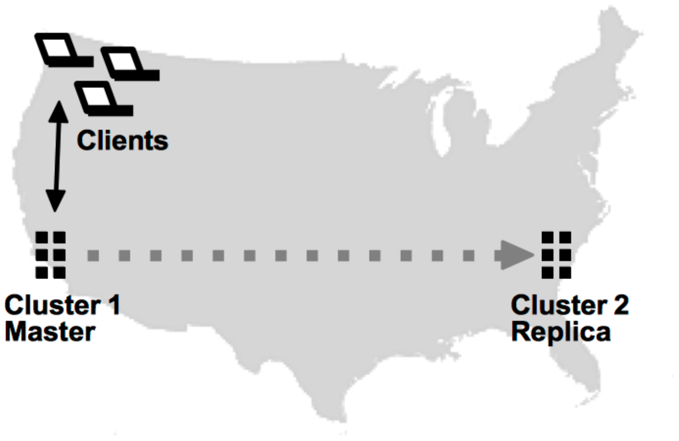


Figure 4.6: Passive database replication [VoltDBDoc]

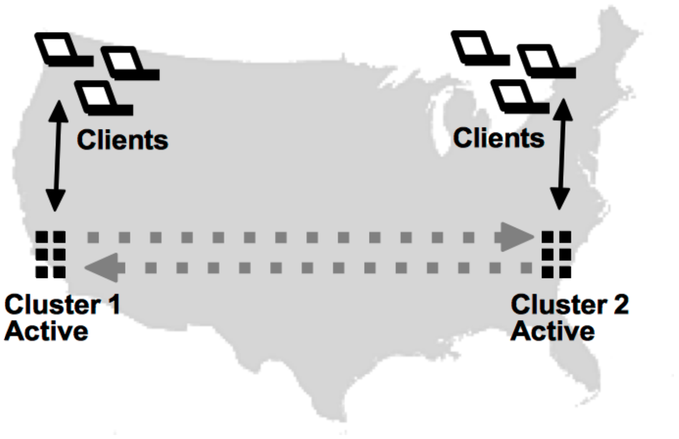


Figure 4.7: Cross datacenter replication [VoltDBDoc]

The test case, which is described in table 2.1 is basic and is performed for each replication method in each test database. The test system in MongoDB consists of a config server and one shard. This shard is a replica set with one primary node and one secondary node. The data inserted into the primary should be automatically transfered to the secondary node. The test

Test case	**Query**	**MongoDB**	**VoltDB**
Replication	Insert 1365 user profiles to the primary database and check if the changes are reflected in replicated databases.	Replica set	Passive replication Cross datacenter

Table 4.16: Replication test

Test case	**Query**	**MongoDB**	**VoltDB**
		Range-based sharding	*Partitioning column*
Single-partitioned procedure	Count the number of users whose ages are between 25 and 35 years old in a given location.	*find* query	Stored procedure

Table 4.17: Single-partitioned procedure test

system for replication test in VoltDB contains two single node clusters, one of which is the primary database, the other acts as a replica.

Single-partitioned procedure

In VoltDB, an user-defined stored procedure can be designed to be routed to and performed only in a specific partition by activating single-partitioning function during creation of the procedure. It should be taken into consideration that all required data has to be on the same partition. Since all data needed reside on the same partition, the search is much more effective. Table 4.17 shows the test query, the sharding method and the reading method of each database. In this test, since the shard key*location* is used, *range-based sharding* is utilized in MongoDB to shard the data. The query is then routed directly to the shard containing only data in this location. Single-partitioned procedure associates a query with a specific partition based on the value of the location. Both databases have in this test the advantages of knowing which partition to reach. The time the databases needed to complete the query is measured.

4.6 Test results

4.6.1 Ecommerce

Performance test

Table 4.18 below shows the results of the performance benchmark of MongoDB and VoltDB in ecommerce scenario. All the time measured are in milliseconds.

Testcase		**MongoDB**		**VoltDB**	
				AdHoc SQL	*Store procedure*
		Single-write	*Bulk-write*		
INSERT	10 records	399 ms	38 ms	371 ms	369 ms
	100 records	2742 ms	79 ms	2749 ms	2656 ms
	1000 records	28664 ms	195 ms	29046 ms	28666 ms
		(Rows returned/time)		*(Rows returned/time)*	
SELECT	Return all products whose prices are between \$25.00 and \$50.00	500 rows/7 ms		500 rows/147 ms	500 rows/171 ms
		(Rows returned/time)		*(Rows returned/time)*	
UPDATE	Apply a reduction of \$5.00 for all the product whose prices higher than \$10.00	545 rows/42 ms		545 rows/24 ms	545 rows/26 ms
		Single-delete	*Bulk-delete*	*Single-delete*	*Bulk-delete*
DELETE	10 records	184 ms	19 ms	240 ms	20 ms
	100 records	1868 ms	21 ms	2285 ms ms	20 ms
	1000 records	18581 ms	32 ms	21479 ms ms	22 ms

Table 4.18: Performance test results

With write operations (INSERT), bulk insertion in MongoDB is shown to outperform single insertion in MongoDB and VoltDB (using AdHoc SQL and stored procedures). The rest, including single insertion in MongoDB and single insertion in VoltDB (using AdHoc SQL and stored procedures), perform roughly equally. This gives MongoDB an advantage if big data sets need to be inserted, especially during high traffic times. Since this is an ecommerce scenario, insertions are much less frequent than read operations, unless the ecommerce shop mentioned is as large as Amazon, in which insertion of reviews and new users happen often. Even so, compared to social networks, insertions are still less often and can be scheduled to optimize performance (e.g. inserting new products at low traffic times).

With SELECT query, MongoDB comes first using *query*. Both VoltDB AdHoc SQL and stored procedure take much longer to return the matches. However, VoltDB with both AdHoc SQL and stored procedure are twice as fast as MongoDB in updating existing data. Both databases perform equally in deleting the whole database. VoltDB, however, takes longer than MongoDB to delete single data records for all three scales of data.

Taking all observations and the test scenario into consideration, MongoDB is in the leading position. For ecommerce, the most important factor out of four CRUD operations is reading data (SELECT). It is the main action performed by customers. INSERT, UPDATE and DELETE operations are less common and if these operations are for product data, they can be scheduled to be performed when the system has less workloads.

Functionality

Joins and aggregation

Test case		MongoDB		VoltDB	
		Application	*Aggregation framework*	*AdHoc SQL*	*Stored procedure*
JOIN WITHOUT AGGREGATION	Return all product reviews from a specific user together with products' names given the name of the user	4 rows/309 ms	4 rows/4 ms	4 rows/24 ms	4 rows/61 ms
	Return all product reviews of a specific product together with the customers' names given the name of the product	5 rows/ 267 ms	5 rows/3 ms	5 rows/28 ms	5 rows/26 ms
JOIN WITH AGGREGATION	Return the average rating and the number of reviews of a specific product, given the product name	1 record((8 reviews/ 4.5 average)/ 88 ms	1 record((8 reviews/ 4.5 average)/ 2ms	1 record((8 reviews/ 4.5 average)/ 21 ms	1 record((8 reviews/ 4.5 average)/ 25 ms
	Return top 20 books with the most reviews	20 rows/17303 ms	20 rows/2ms	20 rows/45ms	20 rows/43ms
	Count the number of products in the category {Book} whose prices are between $25.00 and $50.00	1 row (count = 51)/ 77 ms	1 row (count = 51)/ 2ms	1 row (count = 51)/ 24ms	1 row (count = 51)/ 19ms

Table 4.19: Join test results

According to table 4.19, aggregation framework is the fastest way to perform queries involving joining tables. Coming second are AdHoc SQL and Stored procedure. The slowest is shown to be explicit handling of joins in application codes. Before version 3.2 of MongoDB, left joins were not possible in MongoDB and handling joins in application codes was the only solution. With joins now available in aggregation framework, performance can be largely improved. However, the downside of this approach is the complexity of designing phases of an aggregate pipeline. Identifying phases needed for the queries are harder and not as intuitive as with traditional SQL. Also, one side of a join is fixed with the collection calling the aggregate method. With VoltDB, it is easier to define the joins of tables and it is also possible to have nested joins, though the performance can not compete with that of aggregation framework in MongoDB.

Indexes

Table 4.20 presents the results of index tests. Both of the test databases are capable of creating unique as well as non-unique indexes, single as well as compound indexes. As presented here, MongoDB is the one that shows the clearest signs of benefiting from the performance boost of indexes in both queries. Querying in VoltDB with Adhoc SQL as well as with stored procedures when indexes are used shows inconsistency in performance. The reason for it maybe because the sample data consisting of 1000 products is not big enough for the indexes in VoltDB to be effective. Nevertheless, even without indexes, MongoDB still shows better performance than VoltDB. And since the purpose of indexes is to boost performance of the database, MongoDB is the one to demonstrate the impact of indexes.

Text search

Below in table 4.21 are the results of text search tests.

Test case	Query	MongoDB	VoltDB
Text search	Search for records containing the term "java". *(Exact Match)*	3 rows/8 ms	0 row/ 32 ms
	Search for records based on the wildcard "ava". *(Wildcard)*	0 row/ 6 ms	210 rows/ 38 ms
	Search for records containing any of the following terms: "java", "coffee", "house" *(Terms in a list)*	16 rows/8 ms	34 rows/87 ms
	Search for records containing the term "java", except for records also containing the term "coffee". *(Excluding term)*	10 rows/7 ms	30 rows/36 ms

Table 4.21: Text search test results

With the first query, MongoDB performs text search well, delivering all the exact matches of the term "java" and proves to be case insensitive by delivering matches for "Java" also. Meanwhile, VoltDB searches using LIKE which is case sensitive (only works with "java", not "Java").

The second query is about finding matches based on a wildcard, not an exact term. In MongoDB, no matches were found. The reason is because text search in MongoDB doesn't support wildcards. A possible solution to find documents based on wildcards is using *regex*, which provides "regular expression capabilities for pattern matching strings in queries", according to MongoDB documentation. MongoDoc VoltDB has the advantages here. Using the LIKE operator, it is possible to find matches containing the exact wildcard (no matches containing the wildcard in capital letters because of case sensitivity).

The differences between MongoDB and VoltDB become more apparent with the third query. MongoDB delivers 16 matches, which contains exact matches of the terms "java", "shop" and "coffee". VoltDB instead returns 34 matches using LIKE operator. Since the search in VoltDB is case sensitive, the matches for "Java" were not delivered. However, matches that contains the queried terms as part of another word, e.g. "workshop" are returned. Because of that, it can be concluded that the search in VoltDB, though less accurate due to case sensitivity, has a wider range than the text search functionality in MongoDB, which only looks for exact matches.

In the case of the fourth query, both databases perform equally well on excluding matches containing the term "coffee". However, VoltDB still returns matches containing the word "Coffee". The number of matches in VoltDB is also higher because VoltDB, just as with the third query, also return matches containing words that have the terms "java" or "shop" as part of them.

Test case	**Query**	**MongoDB**		**VoltDB**	
		Without index	*With index*	*Without index*	*With index*
Non-unique single index	Create a non-unique index on the field *price* in table *product.* Perform this query: Return 50 products priced between $25 and $50.	51 rows/14 ms	51 rows/7 ms	51 rows/32 ms (AdHoc SQL) 51 rows/43 ms (Stored procedure)	51 rows/39 ms (AdHoc SQL) 51 rows/38 ms (Stored procedure)
Unique compound index	Create a unique index based on the fields *product_id*, *customer_id* and *review_time* in table *review.* Get the *product_id* of a random product. Perform this query: Return all reviews of a specific product, sorted by the review time in descending order, given the *product_id.*	7 rows/ 10 ms	7 rows/6ms	7 rows/23 ms (AdHoc SQL) 7 rows/24 ms (Stored procedure)	7 rows/24 ms (AdHoc SQL) 7 rows/ 21 ms (Stored procedure)

Table 4.20: Index test results

Performance-wise, the text search in MongoDB is faster, presumably due to the text index, which is part of the text search. On the contrary, VoltDB has no dedicated full text search and has to traverse through every record in the database to find matches for the LIKE operator.

The database performs better in text search tests is MongoDB, in both functionality and performance. It has dedicated text search which is case insensitive. It is indeed helpful, especially during product search since the customers don't need to capitalize any letter just to get correct matches. It is also proven to be much faster when searching for matches. The abilities to receive a list of terms as input and exclude certain terms from the search are also a bonus. Though the text search doesn't support wildcards, this problem can be overcome by combining text search for exact matches with *regex* for pattern matching to have a powerful search tool covering all possible matches.

Transaction

In table 4.22 are the results of the read-only transaction test. As can be seen in the results, aggregation framework in MongoDB delivers the same data ten times faster than transaction in VoltDB. It is indeed helpful while loading a big catalog of products in an online store. A read-only transaction is a transaction without any write operation. Since most transactions in ecommerce are read-only, no existing data is modified and no contraint violations or data inconsistency are caused by these transactions. Performance therefore should be the priority.

Test case		MongoDB (Aggregation framework)	VoltDB (Stored procedure)
Transaction	Deliver 20 products in *book* category which have an average rating of 5.0 and are sorted by price in ascending order	20 rows/2ms	20 rows/22ms

Table 4.22: Transaction test results

Security

MongoDB and VoltDB achieve the desired results for all the queries described in table 4.11. With the first query, the data get updated very quickly for user A in both databases. Granting an user more access to data and resources also works well in both MongoDB and VoltDB for the queries in *role based access control* test case. When an user attempts to access or to perform an action, to which they don't have permission, there are immediate error messages to inform the users about their lack of permissions.

Both MongoDB and VoltDB grant users privileges by assigning roles to them. Users in both databases can have many roles and the roles are cumulative. There are also default roles that can be assigned to users. MongoDB is, however, is more flexible about the level of access control. In MongoDB, not only access to databases can be specified in roles, but also specific actions, such as: find, update, delete can also be defined for collections inside databases. This indeed provides more flexibility in customizing roles. There are not just only read-only or read-write access for the whole database. Users can have different privileges in different collections of the same database, depending on their roles. On the contrary, the choices are more limited in VoltDB. Roles can be created based on default permissions described in table 4.10. These permissions are very basic, specifying only access to all AdHoc SQL, stored procedures of the database. Permissions to specific actions: UPDATE, INSERT, SELECT, DELETE can't be specified for single tables. One way to work around this problem is to grant users read-only access to the whole database, then define stored procedures for specific tasks, such as insert a review of a product, and allow the users'role access to these stored procedure. All of this can be done during the creation of stored procedures. This is also how the security test queries in this thesis are performed in VoltDB.

Compared to traditional SQL databases, VoltDB lacks GRANT ability to give users customized access to the tables in the databases. The workaround with stored procedures can be done, is, however, not optimal. Being able to grant roles different privileges on different collections easily, MongoDB proves to outperform VoltDB in the security tests.

Summary

The purpose of the above tests is to investigate the more suitable for an ecommerce scenario out of the two databases MongoDB and VoltDB. Based on the results, even though it is not

Test case	**Query**	**MongoDB**		**VoltDB**
		Range-based sharding	*Hash sharding*	*Partitioning column*
INSERT	Insert 1365 user profiles (averaging 3 times)	31069 ms	31412 ms	28517 ms
SELECT	Return 1365 user profiles (averaging 3 times)	7 ms	5 ms	64 ms
Find age group	Count the number of users whose ages are between 25 and 35 years old	224 rows/47 ms	224 rows/41 ms	224 rows/25 ms
Friends of friends	Return friends of friends (only ids). The query is performed on 333 user profiles	9890 ms	9691 ms	12994 ms
Friends of friends with profiles	Return friends of friends (with profiles). The query is performed on 333 user profiles	193874 ms	184276 ms	8948 ms
Mutual friends	Return the mutual friends of two users, given their user_id: 23 and 227	2 rows/70 ms	2 rows/90 ms	2 rows/31 ms
Friends in same location	Return the friends of an users, who also live in the same location as the user, given the user_id of the user: 23	7 rows/23 ms	7 rows/85 ms	7 rows/31 ms

Table 4.23: Performance test results

fully capable of joins as VoltDB, MongoDB is still able to perform the common queries that involve joins in an online store. The read operations in MongoDB are faster than in VoltDB, indexes are easy to create and effective. Since the transactions in an online store are mostly read-only ones, MongoDB's lack of ACID support can be tolerated. *Text search* in MongoDB is a powerful tool to query for product data effectively and efficiently, a function which can not be ignored in an online store. Furthermore, there are more security options in MongoDB. The roles in MongoDB can be customized with specific actions, not only on a database level, but also collection level, which is very practical and suits the scenario well, since for different collections customers may different types of access to. It can be concluded that MongoDB is more suitable for the ecommerce scenario, even though ecommerce is a field that is often associated with SQL databases.

4.6.2 Social network

Performance test

Table 4.23 shows the results of performance test in social network scenario. The time measured is in milliseconds.

VoltDB is better than MongoDB at data insertion. It maybe because in the data model of VoltDB for this scenario, the friend list is not stored inside the user profile but in the relationship table, which makes each data record slightly smaller. MongoDB is, however, is better than VoltDB in reading data with both methods of sharding (10 times faster in retrieving 1365 user profiles). Sharding the database using hashed keys makes reading faster in this case because there is no filter in the search, *mongos* performs thus a *broadcast operation* for the query in the whole database. For social networks, the daily number of read operations are more than the number of write operations. Having better performance in reading data put MongoDB in the leading position.

About social network related queries, VoltDB seems to perform better than MongoDB with aggregate function *count*. On the other hand, both *range-based* and *hash* shardings provide the results faster with the query to find ids of friends of friends. VoltDB in turn outperforms in finding profiles of friends of friends and finding mutual friends. The reason for the inconsistencies in performance of MongoDB is because MongoDB is not optimal for joining tables and not capable of having nested joins. The processing of the queries in these 3 cases in MongoDB has to be broken down in smaller parts and be handled in the application code while only one query containing multiple of joins suffice in VoltDB to deliver the desired results. In the case of finding friends of a given user who are also in the same location as that user, *range-based sharding* in MongDB and *partitioning column* in VoltDB display better performance than *hash sharding* in MongoDB. It is because both of them use *location* as shard key, so when a query includes this shard key, only relevant shards are targeted. *Hash sharding* in this case performs searches on all shards.

Although better at single read of user profiles, when more complicated queries, which are essential to social network environment and require multiple levels of joins are involved, MongoDB doesn't perform as well as VoltDB. Overall, VoltDB is the better database in the performance test case.

Functionality

Aggregation
Table 4.24 show the results of the aggregation test for both aggregation methods in MongoDB *aggregation framework* and *map-reduce* as well as the regular SQL query in VoltDB. Of all the three methods, *aggregate framework* proves to be the most efficient one with shortest time to deliver the final result. Compared to it, *map-reduce* in MongoDB is not as efficient, albeit being more flexible with the customization of *map* and *reduce* functions by using custom Javscript. This customization is, however, can become more complex than just using the built-in operators to define phases in an aggregate pipeline. VoltDB also supports *map-reduce* by incorporating *export* functionality to Hadoop, this feature is unfortunately not supported

Test case	**Query**	**MongoDB**		**VoltDB**
		Aggregation framework	*Map Reduce*	*AdHoc SQL*
Aggregation	Return all users who used to study or is studying at the same high school together with their number of friends given the name of the school .	223 rows/3 ms	223 rows/16 ms	223 rows/569 ms

Table 4.24: Aggregation test results

Test case	**Query**	**MongoDB**		**VoltDB**
		Range-based sharding	*Hash sharding*	
Partial index	Create a partial index on users with the feature *high_school*. Return all the users studying at a specific school given the school name.	207 rows/11 ms	207 rows/3 ms	207 rows/27 ms

Table 4.25: Partial index test results

in the test database, which is a community edition, only in enterprise edition. Using only the normal AdHoc SQL to query the data, it is expected to fall far behind both the aggregation methods in MongoDB. Without taking Hadoop's performance into consideration, it can not be denied that the *aggregate framework* in MongoDB delivers impressive performance.

Partial index
In table 4.25 are the results of partial index test. Partial index is a MongoDB feature. Only documents which fulfill certain requirements are indexed, in this case it requires that the attribute *high_school* exists in the indexed document. The results indicate that partial index does help MongoDB search for matches faster, in both cases of sharding, since not all documents need to be traversed through, only a subset having the attribute *high_school*. VoltDB takes longer to perform the search, due to not having partial index. Between both types of sharding in MongoDB, with hash sharding, the results are returned faster, presumably because the data is distributed more equally to all the nodes in the cluster than with range-based sharding, which means there is a balance in the workloads for every node. Partial index aids well in boosting the performance of the whole cluster. The reason for that is because in social networks, not all user profiles have the same attributes. Hence, by applying partial index on an attribute, only a small subset of data need to be traversed through when the query includes this attribute.

Test case	**Query**	**MongoDB**	**VoltDB**
			Stored procedure
Transaction	As an user write a post and publish it (which means the post is saved to the database). After publishing the above post, the page is refreshed and all the posts together with the new post are shown.	4 rows/31 ms	4 rows/42 ms

Table 4.26: Read-write transaction test results

Transaction

Table 4.26 shows slight difference in performance of MongoDB and VoltDB in the read-write transaction test. Despite VoltDB is a little slower than MongoDB, since write actions are involved in this test, stored procedure's ACID nature guarantees data consistency and failure tolerance. VoltDB is thus more suitable in performing transactions which involve write actions.

Distribution

Replication
Replication is tested in both MongoDB and VoltDB by inserting 1365 user profiles into each database and then connect to the replica to check if the data is updated. In MongoDB, only one shard is a replica set with two nodes. After the data being inserted and the primary node is checked for new data, the secondary node is confirmed to have the same amount of data records as the primary node. The following message 4.1 is returned when querying for the status of the secondary node. This proves that the replication was successful and the secondary node is up to date.

```
source: 141.22.32.15:27018
syncedTo: Mon Juli 25 2016 20:10:38 GMT+0200 (CEST)
0 secs (0 hrs) behind the primary
```

Listing 4.1: Replication lag

In VoltDB, both the replication methods *passive* and *cross data center* are tested. The data updates in both methods are very quick. *Passive* replication provides another node in read-only mode with up to date data and it is possible to read data from this node. The replica can replace the master when the master fails. Meanwhile, with *cross datacenter*, no cluster is primary database, read-write are possible in all clusters. Changes from any of them get transfered to the others.

Compare them with each other, *replica set* in MongoDB is the least suitable replication method for social networks since the replica in MongoDB don't provide any access to the data,

Test case	Query	MongoDB	VoltDB
Single-partitioned procedure	Count the number of users whose ages are between 25 and 35 years old in a given location.	80 rows/34ms	80 rows/ 24ms

Table 4.27: Single-partitioned procedure test results

act only as a data backup for the primary node and therefore doesn't contribute to the high availability of the network. Coming second is *passive* replication in VoltDB. By having the data accessible in read-only mode, it increases the high availability of the whole network for read operations. However, because all write operations are performed on the master node, this can leads to bottleneck in performance and even single point of failures. The *cross datacenter* approach maybe the best solution here. The daily amounts of write queries are almost as much as read queries in social networks. Having multiple nodes available for both read and write operations increases high availability, boost performance and helps avoid single point of failures on one master node. It is still possible that changes on one node are not copied to other nodes in case that node fails.

Single-partitioned procedure
Table 4.27 presents the results of the single-partitioned procedure test. VoltDB is slightly faster than MongoDB in delivering the result of the aggregate function *count*. Considering ACID nature of stored procedures, VoltDB seems to be the safer choice when there are write operations, which are very common in social networks (e.g. writing posts, comments). Since the performance of VoltDB also does not fall behind that of MongoDB even when both of them are given the same advantages, VoltDB proves to be the better choice here.

Summary

VoltDB, as seen in performance test result, is better than MongoDB in delivering the result of the aggregate function *count*, data insertion and social network related queries which involve multiple joins. In the performance test, VoltDB also delivers the result of *count* faster than MongoDB. Though not supporting *partial index*, its ACID nature guarantees data consistency and reliability of write actions, which are very common in social networks on a daily basis. It's support of single-partitioned stored procedure also reduces the overall query time by narrowing down the search to a specific partition, which is extremely helpful with data localization. VoltDB also shows better support of replication, having two different methods of replicating data in a cluster. Both of which do not only secure the data, but also increases high availability and performance of the whole system. Without taking aggregation into consideration since Hadoop export functionality can not be tested, VoltDB seems to have more advantages than MongoDB. Considering the fact that social networks are about modeling relations among people, VoltDB is more suitable in modeling social networks because of its nature as a relational database. A test of aggregation methods which involve both methods in MongoDB and Hadoop export functionality in VoltDB should be conducted to make a definite conclusion.

5 Conclusion

This thesis, as stated before, focuses on conducting an application-oriented comparison of MongoDB, a NoSQL representative and VoltDB, a NewSQL representative. The scenarios, which are chosen for testing, are ecommerce and social network. Both of them are typical scenarios in daily lives. One is a typical example usually associated with traditional SQL databases while the other is operated in a distributing environment, which is NoSQL databases' strength.

Over the course of this thesis, testing criteria and testing queries are tailored to suit the targeted scenario. The results, including the performance time, are documented and used to identify the more suitable database for each test scenario.

In ecommerce test scenario, MongoDB proves to be a better candidate with adequate support for joins, effective indexes, better performance in reading operations, flexibility in customizing user roles on multiple levels and in explicit details. It's support for full text search fits well into an ecommerce scenario also. VoltDB, even though being a relational database with fully ACID transaction and join support, is less suitable performance-wise. Moreover, since transactions in ecommerce are mostly read-only, data consistency matters less.

In social network test scenario, VoltDB seems to be in a leading position. Based on the fact that social networks center around relations, a relational database is more suitable for modeling data. Since the common social network related queries involve multiple nested joins, VoltDB's fully support of join simplifies the creation and conduction of queries. MongoDB, instead, due to it's limited support of joins, needs to break down the complex queries and handle them explicitly in application codes, which raises the complexity of designing queries and reduces performance. Replicating data in VoltDB is not just about securing data. By having more nodes with the same data and open access to them, data replication in VoltDB also help increase high availability and boost performance of the whole system. Compared to that, replication in MongoDB only exist as backup for the primary database and only usable when the primary database fails.

Although this thesis attempts to cover as many aspects as possible, there are still opening matters to be investigated. One of them is VoltDB's support of Hadoop, which is supposedly increases performance noticeably. The scenarios tested here are also very limited. Other aspects, such as delivery and payment in ecommerce as well as likes in social networks should also be used to test MongoDB and VoltDB before making conclusions.

Bibliography

[Aboutorabi u. a. 2015] Aboutorabi, S. H. ; Rezapour, M. ; Moradi, M. ; Ghadiri, N.: Performance evaluation of SQL and MongoDB databases for big e-commerce data. In: *Computer Science and Software Engineering (CSSE), 2015 International Symposium on*, Aug 2015, S. 1–7

[ArangoDB] ArangoDB: Benchmark: PostgreSQL, MongoDB, Neo4j, OrientDB and ArangoDB, URL https://www.arangodb.com/2015/10/benchmark-postgresql-mongodb-arangodb/

[Bain 2009] Bain, Tony: Is the Relational Database Doomed? (2009). – URL http://readwrite.com/2009/02/12/is-the-relational-database-doomed/#awesm=~ookpfnVjGmGXHJ

[Basel Kayyali und Kuiken 2013] Basel Kayyali, David K. ; Kuiken, Steve V.: How big data is shaping US health care. (2013). – URL http://www.mckinsey.com/industries/healthcare-systems-and-services/our-insights/how-big-data-is-shaping-us-health-care

[Bernstein 2014] Bernstein, D.: Today's Tidbit: VoltDB, May 2014, S. 90–92. – ISSN 2325-6095

[bigdatanerd 2011] bigdatanerd: WHY NOSQL-PART 1âCAP THEOREM, URL https://bigdatanerd.wordpress.com/2011/12/08/why-nosql-part-1-cap-theorem/, December 2011

[Boicea u. a. 2012] Boicea, A. ; Radulescu, F. ; Agapin, L. I.: MongoDB vs Oracle – Database Comparison. In: *Emerging Intelligent Data and Web Technologies (EIDWT), 2012 Third International Conference on*, Sept 2012, S. 330–335

[Brewer 2012] Brewer, Eric: CAP Twelve Years Later: How the "Rules" Have Changed. (2012). – URL http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

[Cox und Ellsworth 1997] Cox, Michael ; Ellsworth, David: Application-controlled demand paging for out-of-core visualization. (1997)

[Date und Darwen 1997] Date, C. J. ; Darwen, H.: (1997). ISBN 0-201-96426-0

[Doshi u. a. 2013] Doshi, K. A. ; Zhong, T. ; Lu, Z. ; Tang, X. ; Lou, T. ; Deng, G.: Blending SQL and NewSQL Approaches: Reference Architectures for Enterprise Big Data Challenges. In: *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2013 International Conference on*, Oct 2013, S. 163–170

[Eifrem und Rathle 2013] Eifrem, Emil ; Rathle, Philip: The most important part of Facebook Graph Search is âGraphâ. (2013). – URL http://neo4j.com/blog/why-the-most-important-part-of-facebook-graph-search-is-graph/

[Gantz und Reinsel 2010] Gantz, John ; Reinsel, David: The Digital Universe Decade Are You Ready? (2010). – URL http://www.emc.com/collateral/analyst-reports/idc-digital-universe-are-you-ready.pdf

[Gen] Gen: MySQL vs. MongoDB: The Pros and Cons When Building a Social Network, URL https://www.morpheusdata.com/blog/2015-04-01-mysql-vs-mongodb-the-pros-and-cons-when-building-a-social

[Gunelius 2014] Gunelius, Susan: The Data Explosion in 2014 Minute by Minute â Infographic. (2014). – URL http://aci.info/2014/07/12/the-data-explosion-in-2014-minute-by-minute-infographic/

[ISTC] ISTC: Benchmarking Graph Databases, URL http://istc-bigdata.org/index.php/benchmarking-graph-databases/

[J. McAuley 2015] J. McAuley, J. L.: *Inferring networks of substitutable and complementary products*. 2015. – URL http://cseweb.ucsd.edu/~jmcauley/pdfs/kdd15.pdf

[Laney 2001] Laney, Doug: 3D Data Management: Controlling Data Volume, Velocity and Variety. (2001)

[Li und Manoharan 2013] Li, Y. ; Manoharan, S.: A performance comparison of SQL and NoSQL databases. In: *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, Aug 2013, S. 15–19. – ISSN 1555-5798

[Mathew und Kumar 2015] Mathew, A. B. ; Kumar, S. D. M.: Analysis of data management and query handling in social networks using NoSQL databases. In: *Advances in Computing, Communications and Informatics (ICACCI), 2015 International Conference on*, Aug 2015, S. 800–806

[Microsoft] Microsoft: Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence. . – URL https://msdn.microsoft.com/en-us/library/dn313285.aspx#sec7

[MongoDBBlog] MongoDBBlog: *Joins and Other Aggregation Enhancements Coming in MongoDB 3.2*. – URL https://www.mongodb.com/blog/post/joins-and-other-aggregation-enhancements-coming-in-mongodb-3-2-part-

[MongoDBDoc 2016] MongoDBDoc: Sharding Introduction, URL https://docs.mongodb.com/manual/core/sharding-introduction/, 2016

[MongoDoc] MongoDoc: SQL to MongoDB Mapping Chart, URL https://docs.mongodb.com/manual/reference/sql-comparison/

[Morgan 2015] Morgan, Andrew: Joins and Other Aggregation Enhancements Coming in MongoDB 3.2, URL https://www.mongodb.com/blog/post/joins-and-other-aggregation-enhancements-coming-in-mongodb-3-2-part-
2015

[Parker u. a. 2013] Parker, Zachary ; Poe, Scott ; Vrbsky, Susan V.: Comparing NoSQL MongoDB to an SQL DB. In: *Proceedings of the 51st ACM Southeast Conference*. New York, NY, USA : ACM, 2013 (ACMSE '13), S. 5:1–5:6. – URL http://doi.acm.org/10.1145/2498328.2500047. – ISBN 978-1-4503-1901-0

[Standford] Standford: *Standford Snapshot*. – URL https://snap.stanford.edu/data/

[Stonebraker u. a. 2007] Stonebraker, Michael ; Madden, Samuel ; Abadi, Daniel J. ; Harizopoulos, Stavros ; Hachem, Nabil ; Helland, Pat: The End of an Architectural Era: (It's Time for a Complete Rewrite). In: *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB Endowment, 2007 (VLDB '07), S. 1150–1160. – URL http://dl.acm.org/citation.cfm?id=1325851.1325981. – ISBN 978-1-59593-649-3

[VoltDBDoc] VoltDBDoc: VoltDB Documentation, URL https://docs.voltdb.com

[Wall 2014] Wall, Matthew: Ebola: Can big data analytics help contain its spread? (2014). – URL http://www.bbc.com/news/business-29617831

[Wikipedia] Wikipedia: *Social network*. – URL https://en.wikipedia.org/wiki/Social_network

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 8. August 2016 Ngoc Huyen Nguyen