A performance comparison of NoSQL and SQL databases for different scales of ecommerce systems

Wenbin Shen (17002855)

Master of Computer and Information Sciences

Auckland University of Technology

Table of Contents

Abstract ... 4

Introduction ... 4

Literature Review ... 6

- Ecommerce ... 7
- Database Management Systems ... 15
- Database Selection ... 23
- Relational databases ... 24
- NoSQL ... 29
- Performance Testing ... 37
- Literature review summary ... 44

Research Questions ... 45

- Main research question ... 45
- Sub-questions ... 45
- Hypotheses ... 45

Research Design ... 46

- Selecting Databases ... 46
- Designing Database Schema ... 46
- Importing Data ... 50
- Designing Use Cases ... 52
- Setup Experimental Environment ... 53
- Designing Experiments ... 54

Experiment results and findings ... 62

- Product search results and findings ... 63
- Order placement results and findings ... 73
- Order update results and findings ... 77
- Order deletion results and findings ... 81
- Summary ... 85

Discussion ... 87

- Single-thread ... 87
- Multi-threads ... 89
- Summary ... 90

Conclusion ... 91

Limitations 92
Future research 92
References 93
Appendix 104

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgments), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Signature: Date: 4/3/2022

Abstract

Customers have changed their shopping behaviour from shopping in physical stores to shopping on virtual online platforms over the last decades especially since covid-19 lockdowns. Correspondently, this change in shopping behaviours has made it essential for businesses owners to improve their ecommerce platforms to become robust and scalable, while database enhancement is the most critical part of this robust platform. With the emerging technologies of NoSQL, and various database options on the market, ecommerce system developers would wonder whether NoSQL is a better option for their platforms. To help ecommerce system developers make better database decisions, this research conducted 9 use case tests (5 single-thread tests, 4 multiple- thread tests) with CRUD (Create, Read, Update, Delete) operations to compare the performance of SQL (PostgreSQL) and NoSQL (MongoDB) databases with real ecommerce data exported from Kaggle (Kaggle, n.d.-a). In these 9 tests, PostgreSQL outperformed MongoDB in nearly 7 tests (3 single-thread tests, 4 multiple-thread tests), while MongoDB performed better in insert and delete operations with single thread scenarios. Therefore, this research found (within its single host design constraints) that PostgreSQL is a better option for ecommerce platforms where large amounts of concurrent requests happen frequently. The author also suspects that the nature of the ecommerce data model, which is more relational, determines the result that SQL performs better than NoSQL in ecommerce scenarios.

Introduction

Due to the rapid development of internet and smart / mobile devices, ecommerce platforms have become increasingly popular and important in our daily life (Aatish, 2017; Al-Tit, 2020). Additionally, Covid-19 pandemic and lockdowns have changed customer shopping behaviour and they tend to or have to purchase products from online shops, and this change enforces many business owners to shift their sale activities from face to face to a virtual online platform also known as ecommerce systems. However, these ecommerce systems tend to grow speedily in many cases, and they tend to generate large amounts of data including inventory data, business transactions and system logs, so the data storage and processing requirements exceed the capacity of a normal SQL database (Liu, 2015). Ecommerce systems developed before NoSQL are now

questioning whether it is cheaper to replace the existing SQL database with a NoSQL option, or whether it is easier to simply scale vertically with more expensive hardware.

In the current ecommerce market, most ecommerce systems are still using SQL databases as they have a stable capability of processing transactions with ACID (Atomicity, Consistency, Isolation and Durability) compliance (cf. P. 27) and executing complex queries with the powerful SQL syntax. However, it is a fact that we are also in the big data era where various types of data, mostly unstructured, are generated every second 24/7, and high scalability is an essential characteristic that an ecommerce system requires. NoSQL databases can easily cope with a huge amount and various types of data because of their special storage structures.

Therefore, when developing a brand-new ecommerce system, many developers will have to choose from a relational database like PostgreSQL, Oracle and MySQL, or a NoSQL or NewSQL database like MongoDB, CouchDB and Redis (Davis, 2016). Unfortunately, the huge number of database management system options and underlying technologies have made it too confusing for developers to decide which option is the best for their growing ecommerce data centre (Makris, Tserpes, Spiliopoulos, & Anagnostopoulos, 2019). Hence, the research community plays a key role in testing and clarifying the pros and cons of adopting different database systems in the ecommerce field, as these systems normally face scalability and performance issues earlier than other areas.

With the popularity of NoSQL being applied in big data and real-time applications, many developers are wondering whether NoSQL is the answer for their ecommerce systems. On the other hand, many researchers argued that NoSQL databases sacrificed ACID compliance to achieve high performance and scalability (DeCandia et al., 2007; Pokorny, 2013) and suggested people to evaluate the importance of consistency before making database decisions. Hence, a decision between SQL and NoSQL sounds like a choice between consistency and scalability. Instead of following the real time consistency required in ACID compliance, NoSQL databases pursue eventual consistency defined in the BASE consistency model (*Basically Available, Soft state and Eventual Consistency* - cf. p. 34), which means the system will be consistent after all

processes been executed. However, there are no clear boundaries between SQL and NoSQL, as it is becoming a trend that NoSQL databases can achieve strong consistency while SQL starts to embrace some NoSQL characteristics. For example, MongoDB can achieve strong consistency with no partition using its default configuration to read from the primary (Stack Overflow, n.d), and PostgreSQL 9.3 can be turned into a NoSQL by storing JSON files with constraints on fields data (Mohammed, 2015a). Therefore, a database decision cannot be made as simple as "choosing SQL for consistency or choosing NoSQL for scalability".

Additionally, the complexity of an ecommerce data model made it unique and difficult to simply state which database is the best for ecommerce systems. There are small scale ecommerce systems owned by a single small business where large numbers of concurrent requests may be rare on these platforms, so a traditional SQL may be a simpler and better choice, while there are also platforms who do not own the products sold in the system but simply provide a place for buyers and sellers to conduct purchase activities. These platforms tend to involve a larger number of users and may have cases where many concurrent requests are submitted to the database in peak hours, so databases that can perform stably and efficiently would be a better option.

To test the performance and scalability of NoSQL and SQL databases, this research will execute both single thread tests and multiple threads tests using CRUD operations in PostgreSQL and MongoDB. It is important to test multi-thread scenarios as large scale ecommerce platforms tend to have concurrent requests more often, which can reach a peak in special seasons like double 11 for Taobao and prime day for Amazon. In this research, there will be in total 9 tests conducted using real ecommerce data extracted from Olist (a Brazil marketplace system), including 5 single thread tests: insert, search, aggregate search, update and delete operations, and 4 multiple thread tests: insert, search, update and delete operations. In single thread tests, different size of records: 100, 1,000, 10,000 and 100,000 will be used to represent different scope of ecommerce systems, while in multiple thread tests, there will be sizes of 60, 120, 240, 480 and 960 concurrent users.

Literature Review

In this literature review, the author will firstly review the history and different types of ecommerce

systems to identify special testing requirements or things that are notable for this research, and this will be followed by a review on various databases on the market to understand the market occupation of SQL and NoSQL databases which will help the author to select specific databases to represent each database type for the performance and scalability test. Additionally, advantages and disadvantages of using SQL and NoSQL databases will be analysed to help developers make database decisions and support the author define and refine use cases in research design. At the end, tools, benchmarks, and methodologies used in similar research projects will be analysed to format the essential part of the experiment design.

Ecommerce

Electronic commerce systems, also known as ecommerce systems, are a platform for product providers and seekers to buy and sell over the internet using digital devices like computer, tablet and phone. This new form of sales activity has caused a strategic challenge for normal retail and consumer services significantly (Reynolds, 2000). Therefore, it is important to review how ecommerce system developed, what types of ecommerce platforms there are and why and how they have become so popular that scalability becomes a major issue.

History of ecommerce Systems

The ecommerce history started from IBM's Online Transaction Processing (OLTP) to process payment transactions in real-time in the 1960s. This technology was firstly used by different American travel agencies to purchase airline tickets at the same time (Woodford, 2005, p. 100). Before Tim Berners-Lee developed the first World Wide Web server in 1989 (Berners-Lee, Dimitroyannis, Mallinckrodt, & McKay, 1994), Michael Aldrich (pioneer of online shopping) initiated the first teleshopping B2B system in 1981 and expanded it to B2C in 1984 (Al-Tit, 2020). Later when internet and online banking were used for commercial purposes in the 1990s (Kenton, 2021), big ecommerce systems like Amazon and eBay were established, followed by Alibaba, Taobo and Tmall in the 2000's. At that time, managing user trust without social presence was a major concern to many business owners (Gefen & Straub, 2003). The consequent success of ecommerce was partly contributed by the bloom of internet usage which has increased 1331.9% from 2000 to 2021 according to Internet World Stats (2021).

Types of ecommerce systems

There are four major types of ecommerce systems: B2B, B2C, C2C and C2B. (Jiang, 2020) illustrated these four types in the following image.

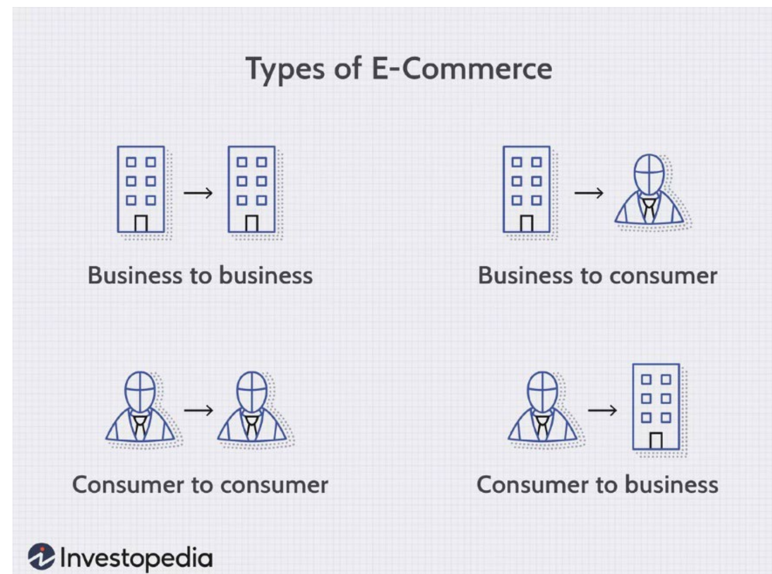

Figure 1: Types of ecommerce systems (Jiang, 2020)

B2C

Business-to-Consumer, also called B2C covers the process of businesses selling products to individual consumers who are also the end user of the product. Investopedia summarised five types of B2C models:

1. Direct sellers

These are the business owners who build their own website to sell products to customers directly. This normally involves a fee for website development and maintenance, and this can be very expensive. Therefore, many small businesses tend to skip this option.

2. Online intermediaries

These are the platforms like Tmall and JD, who do not own the products on their websites, and they are just a media to connect businesses and individual customers.

3. Advertising-based B2C

This type of platforms normally collect many free but attractive contents to achieve higher website traffic, so that they can make benefit from selling advertisements on their site.

4. Community-based

Facebook falls into this classification, which builds an online community for businesses to promote products directly to targeted customers.

5. Fee-based

This includes platforms like Netflix where customers need to pay to access their contents.
Among the five types of B2C models discussed above, “online intermediaries” is the type who attracts the biggest volume of website traffic and involves more complicated user operations like adding products into shopping cart, communicating with individual business owners, and paying via different channels. Therefore, these platforms are the first ones suffering from growing data volumes and motivate the development of new concept databases other than relational databases. Hence, when looking for data samples for the performance and scalability tests in the research, the author will seek “online intermediaries” platforms.

B2B

In people’s general understanding, B2C platforms should have a bigger profit than B2B, as they target on a much bigger market. However, according to ecommerceDB.com (2019), the B2B ecommerce market was valued over six times the B2C market and reached US$12.2 trillion in 2019. One of the biggest and most successful B2B platforms that has emerged in recent years is Shopify, who provides a platform for businesses to build their own ecommerce systems. Based on the statistics Shopify published on their website, there are more than 1.7 million businesses chose Shopify to build their online shops (Shopify, n.d.). In 2014, Shopify launched Shopify Plus to offer more complete integration and customisation services for large enterprises like Nestle, Leese, Pepsi and Bombas. For the twelve months ending March 2021, Shopify revenue reached 3.448 billion, doubling the figure in March 2020. Other B2B successful examples like Alibaba and Made-in- China are all making good profit from their customers who are

businesses, largely because they are richer than individual customers and more willing to pay thousands a month for a good service.

Among the B2B examples discussed above, Alibaba has the largest volume of website traffic of 98.85 million visits in May 2021 (Similarweb, n.d.-a), while B2C platforms like Tmall and JD have a higher traffic of 118.18m and 186.57m separately. Therefore, from the need of a better performance database, B2C platforms are motivated to embrace new concept database management systems.

C2C

Customer-to-customer (C2C) websites normally work as an intermediary to match private sellers with buyers, normally without quality checking. Trademe is a very successful C2C platform not only in New Zealand but also world wide, ranking #1 globally in category “Ecommerce and shopping: Auctions” on Similarweb (n.d.-b) with a monthly traffic of 20 million visits (96% from New Zealand where population is only about 5 million). Therefore, nearly everyone in New Zealand uses Trademe more than four times a month. Another successful C2C system is Facebook Marketplace focusing on sales in local neighbours, which has become “the fifth most popular online marketplace” following Amazon, eBay, Etsy and Walmart (Barkho, 2021). Unlike B2C platforms, C2C websites have to deal with mutual trust among purchasers and providers, so review functions are very important to these platforms.

C2B

The most common C2B adoption is business review platforms where consumers write reviews on a product or service from a business. The most successful C2B example in New Zealand is NoCowboys with 112,400 reviews online. Comparing with other ecommerce models, C2B has a much smaller traffic volume.

After analysing the above mentioned four ecommerce modules, it is noticed that B2C and C2C are the ones tending to have a higher website traffic, with some of them like Amazon who can hit a peak of 2.77 billion visits in January 2021 Similarweb (n.d.-c). Additionally, B2C platforms

normally involve more complicated operations like adding/deleting products to/from cart and updating the quantity of a product. Hence, B2C systems would be the first group which will encounter database issues caused by increasing data volume and scalability. Therefore, when designing this research, the author will focus on building a B2C database schema with data samples from a B2C platform.

Popularity and importance of ecommerce systems

According to Zaied (2012), all business transactions that are completed via the internet are defined as e-commerce, and due to the development of internet, ecommerce has become increasingly popular and common in our daily life (Al-Tit, 2020). Furthermore, the recently boosted usage of smartphones has made ecommerce much more popular in today's digital world (Aatish, 2017), even though, shop-based retail is still the most widespread and profitable (Clement, 2020).

Comparing to business transactions conducted without internet, ecommerce via internet is much quicker and accurate (Sebora, Lee, & Sukasame, 2009), and more importantly, ecommerce has made it much easier for entrepreneurs to develop and grow their global markets (Mullane, Peters, & Bullington, 2001). Many shop-based retailers observed the importance of ecommerce platforms and developed their online shopping options for customers. Wal-Mart is a good example, who has both online and offline outlets, while one of the world's most well-known B2C platform Taobao is web-based only (Trade on Taobao, n.d.).

China is a good example to demonstrate how speedily ecommerce has developed in the past decade. From Chart 1 generated from Ma (2020a)'s research, it is evident that the number of online shoppers has grown exponentially from 2009's 108 million to 2020's 749.39 million, which has made China the world's second-largest ecommerce market after America. Behind these online shoppers are ecommerce transactions, and according to Ma (2020b)'s further research, China's ecommerce transactions reached approximately 9.9 trillion CNY in 2019, which is roughly 2 trillion NZD. On the other hand, according to the ecommerce Report 2021 release d by Statista (2021), China occupied US$1,343.5 billion in online sales in 2020 when

the global ecommerce sales reached 4.28 trillion US dollars, and the report predicted that China would continue dominating the ecommerce market through 2025. This report also revealed a trend that the purchasing power has started shifting from US and Europe to China and Asian markets, because of the boost of internet development and an easier access to mobile devices in those countries.

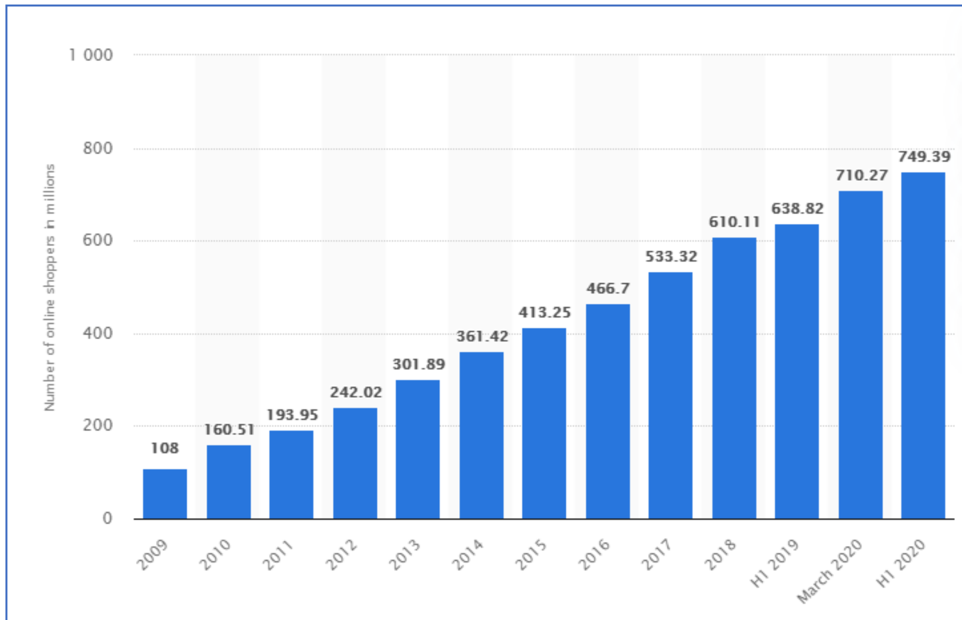

Chart 1: Number of online shoppers in China from 2009 - 2020 (Ma, 2020a)

On the other side of the world, USA's ecommerce market is similarly becoming popular. In 2010, only 4 percent share of total retail sales in USA were ecommerce sales, while this number climbed to 11.2 percent in 2019 (Clement, 2020). Based on the analysis done by Statista (2021), US is the second largest ecommerce market with sales of US$537.7 billion and this number is expected to grow to $723.6 billion by 2025. On the other hand, Europe occupied the third largest ecommerce market of US$460.5 billion with an expectation of US$655.6 billion by 2025.

Covid-19 and ecommerce

Additionally, due to the Covid-19 pandemic in 2020, people tend to avoid physical shops and

purchase essential products online. Chart 2 from Rakuten Intelligence shows people spent 2.5 times more on grocery online shopping in the second week of March when a lot of countries went into lockdown (Meyer, 2020).

Chart 2: Daily ecommerce spending (Meyer, 2020)

To summarise the above findings, an ecommerce system is playing a vital role in the business world, especially during the Covid-19 pandemic time. Therefore, many business owners have started thinking whether they need to develop an ecommerce system and transfer their sales activities to online, as shop-based businesses can hardly survive from lockdowns. On the other hand, for businesses who already have ecommerce systems, they may start thinking whether to replace the traditional database with a NoSQL database to manage the rapidly growing volume of data. However, when adopting ecommerce systems, many entrepreneurs' decisions have been influenced by technology factors like information infrastructure cost, reliability and technology vendors' support (Al-Tit, 2020). Furthermore, among these technical factors, storing and processing a large volume of data using a scalable and practical database is a critical factor that many businesses would not ignore (Liu, 2015).

Businesses of different sizes may have different requirements of data storage and processing, so there may not be "one solution for all problems". Additionally, a small-scale ecommerce system may unexpectedly become very popular in the near future (e.g., Covid-19 increases the demand of grocery online shopping). This also means the relational database which was

suitable to store and process a relatively smaller amount of data may not suit the unexpectedly increasing data volume. Hence, the scalability of a relational database is also an element that should not be ignored when selecting databases for ecommerce systems, as hardly anyone can predict how big the data volume will be in future. On the other hand, if the business did not make "the right choice" at the beginning, it can be very costly to correct their mistakes in the future.

Ecommerce Traffic and Scalability

According to Clement (2021), online shopping platforms like Amazon and eBay have very busy traffic every day when Amazon's average monthly visitors can reach 3.68 billion, more than ten times of USA population of 328.2 million (Data Commons, 2019). Therefore, it becomes crucial for ecommerce platforms to have the ability to scale the site to serve unexpected heavy volumes. Taking Amazon's prime day crash in 2018 as an example: because it failed to manage the surge in volume online and on mobile phones, the system crashed only minutes into the sale: the system was super slow; users could not checkout or even login to the system. Sana Roessel (2018) estimated that this outage costed Amazon $1 million per minute as it happened at the peak period of the prime day and lasted for more than 1 hour. Therefore, heavy traffic is a double-edged sword, which creates profit for the ecommerce system on one side but also produces scalability issues.

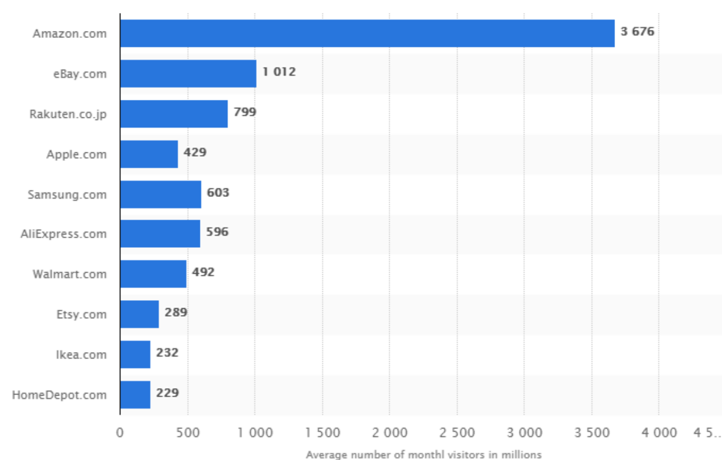

Figure 2: Most popular online retail websites worldwide in 2020, by average monthly traffic

From this Amazon's website incident, businesses realised the importance of the scalability of their systems. It should not only cope with daily volumes but should also be prepared for unexpected challenges in future, as Everts (2016) indicated that the majority of users would abandon a website if it took longer than 3 seconds to load. However, performance of a platform is affected by several key parts including web servers, database servers and hardware (Menascé, 2002). For this research, we will focus on the study of database scalability, as a database is the most essential part of a website, and its performance will heavily affect user experience.

For a database scalability test, it is difficult to decide the number of concurrent users. Taking Amazon as an example, as shown in the above graphic, it has 3.6 billion visitors a month, which means 120 million a day, 500,000 per hour and around 80,000 per minute. The least popular platform listed in the above graphic is HomeDepot, which has 229 million users per month, on average 7.6 million per day, 5,300 users per minute.

Database Management Systems

Database management systems have a relatively short history, started in the 1960s from the Generalized Update Access Method (GUAM) developed by North American Aviation which is now called Rockwell International (Sumathi & Esakkirajan, 2007, p. 5). GUAM was shortly supplanted by other database models like hierarchical, network and relational databases.

Hierarchical databases

In 1960s, IBM joined GUAM but developed it into Information Management System (IMS), a hierarchical database where "the elements of the structure have only one-to-many relationships with one another", as illustrated in the following figure (Batra, 2018; Homan & Kovacs, 2009). It was widely used in the early mainframe computer era but soon replaced by relational databases due to its inflexibility. In hierarchical model, child records can only have one parent, so it does not support many-to-one or many-to-many relationships, and because of its top-down data structure, searching a hierarchical database is time consuming (OmniSci, n.d.). Today, this technology is still in use to store files and geographic information, while some telecommunication and banking

systems also use hierarchical databases because of its high performance in traversing a database and the fact that adding or deleting information would not affect the entirety of the database (EDUCBA, n.d.; OmniSci, n.d.). However, it should be noted that this model can only be used when these two conditions are satisfied:

- Data is stored in a hierarchical patten.
- Data must be accessed by a single path.

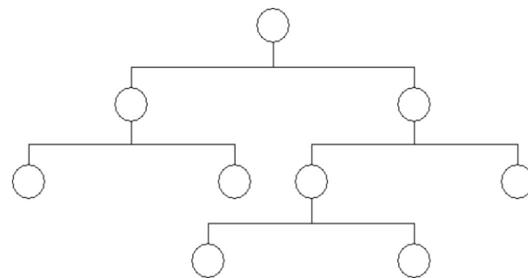

Figure 3: Simple Hierarchical Database Model (Homan & Kovacs, 2009)

Network databases

Integrated Data Store (IDS), a network database management system was developed by Charles Bachman in 1969 to enhance the hierarchical database model (Chand, 2019). In network databases, relationships are managed in a net-like form where one element can point to multiple elements and be pointed to by many other elements (TechOpedia, n.d.), as shown in the following figure. It was believed to be a progression of hierarchical databases to solve its flexibility issues to allow each child to have multiple parents (Maria DB, n.d.). This database model was common on mainframe and minicomputers in the 1970s until it was supplanted by relational databases in the 1980s (Wikipedia, n.d.-a), mainly because of its complex structure that makes it difficult to modify the database after creation (Fandom, n.d). However, its high performance was recognised in a case study performed by RAIMA (n.d.) by SQL developers, comparing a network database model and relational model using three tables: ARTIST, ALBUM and SONGS, as shown in the following table. Therefore, for occasions where faster access is prioritised and data structure is not changed frequently, a network model can be an option (RAIMA, n.d.).

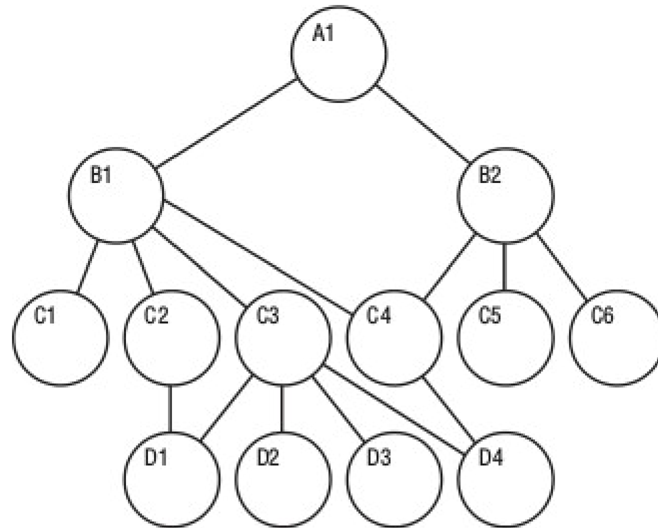

Figure 4: Network Database Model (Maria DB, n.d.)

	Hardware			
	x86 desktop (34,000 records)		ARM7 consumer device (1,776 records)	
	Relational	Network	Relational	Network
Records Inserted	81.62 seconds	29.07 seconds	193.88 seconds	33.60 seconds
Records Updated	103.28 seconds	15.28 seconds	N/A	N/A
Records Deleted	88.16 seconds	17.03 seconds	N/A	N/A
Records Selected	15.81 seconds	0.28 seconds	1.25 seconds	0.0012 seconds

Table 1: Benchmark results from relational and network models on x86 and ARM7 systems (RAIMA, n.d.)

Relational Database

In our daily life, we are using relational databases consciously or unconsciously from swiping a card to purchase groceries in a supermarket, to making appointments with doctors, as most business data are stored in relational databases, a most successful invention of software products in the 1980s (Wade & Chamberlin, 2012).

The idea of Relational Database was first initiated by Codd (1970) from IBM Research Laboratory and in his article published in Communications of the ACM in 1970, he called it “a relational model of data for large shared data bank”. In the late 1970s, System R was invented by IBM to prove that the concept of relational database could be built and work efficiently (Sumathi &

Esakkirajan, 2007, p. 65). In 1985, to define "Relational", Codd (1985) formulated twelve rules that are required for a relational database management system. However according to Date (2005), hardly any commercial database vendors conform to all twelve rules, but they generally follow two concepts: data are presented in tabular format with rows and columns; Relational operators are used to manipulate the data. There were some arguments that only DBMSs that follow all Codd's 12 rules are truly-relational databases, while others are pseudo-relational databases (Wikipedia, n.d.-c). However, it is common these days that people accept a database as a relational database if it can be queried using SQL, a non-procedural language used by non-programmers to perform database administrative tasks (Ramakrsihnan, Donjerkovic, Ranganathan, Beyer, & Krishnaprasad, 1998; Wade & Chamberlin, 2012).

The first relational database management system was created by Honeywell in 1976 and was called Multics Relational Data Store (MRDS), with only basic commands like the equivalent SELECT and UPDATE in SQL (Weeldreyer & Friesen, 2001). In late 1970s and early 1980s, commercial companies IBM and Relational Software (now well known as Oracle Corporation) launched their first relational database management systems: BS12 and Oracle (Darwen, 1996; Oracle, 2007). Later Semantic Data Modelling emerged to closely represent the "real world" in DBMSs, including Codd's RM/T and RM/V2 (Sumathi & Esakkirajan, 2007, p. 5).

To explain the history of database management systems clearly, Sumathi and Esakkirajan (2007, p. 6) created this list:

- Flat files – 1960s–1980s
- Hierarchical – 1970s–1990s
- Network – 1970s–1990s
- Relational – 1980s–present
- Object-oriented – 1990s–present
- Object-relational – 1990s–present
- Data warehousing – 1980s–present
- Web-enabled – 1990s–present

Nowadays, the database management systems market is still dominated by relational databases with NoSQL and NewSQL taking some of the shares (Simsek, 2019).

Databases on the market

In this online shopping era, the data volume of many ecommerce platforms has expanded enormously, which increases the demand of high scalability and better data processing databases. On the current database market, there are mainly three types of databases: traditional relational databases, NoSQL and NewSQL. Relational databases were invented in the earlier era when hardware and software technologies were very different from today, so it is not surprising to see these traditional databases facing problems to process big data which requires higher performance and scale requirements (Grolinger, Higashino, Tiwari, & Capretz, 2013). NoSQL and NewSQL databases have emerged to handle these issues arising from processing big volumes of data.

Unlike relational databases which focus on relationships among different datasets, NoSQL and NewSQL weaken relationships and realise the benefits of using Document Databases and Key-Value Stores (Liu, 2015). However, because the number of databases available on the market, it is very difficult for developers to decide which one is suitable for their ecommerce project.

Back to 2013 when there were 204 different database systems on the market, 77 of them (38%) were relational database management systems (RDBMSs) (Andlinger, 2013). In 2021, the total number of DBMSs has reached to 373, and 39% of them are RDBMSs (DB-Engines, 2021c). However, the popularity of RDBMSs has dropped 15.6% from 90.8% to 72.8% (DB-Engines, 2021c).

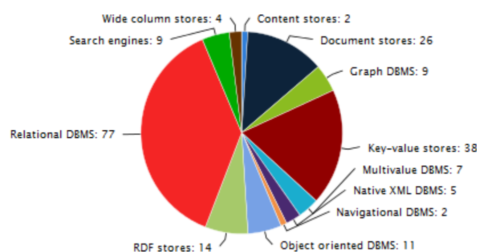

Chart 3: Number of DBMSs 2013

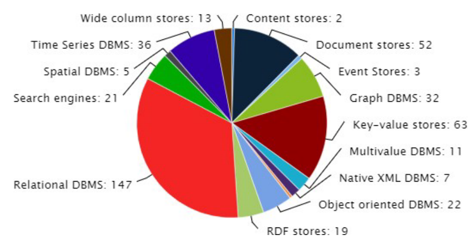

Chart 4: Number of DBMSs 2021

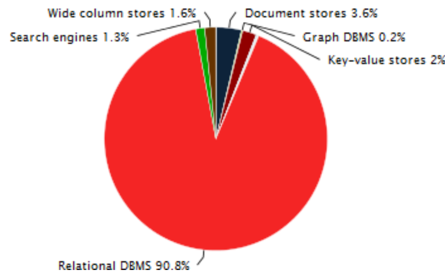

Chart 5: Ranking scores per category 2013

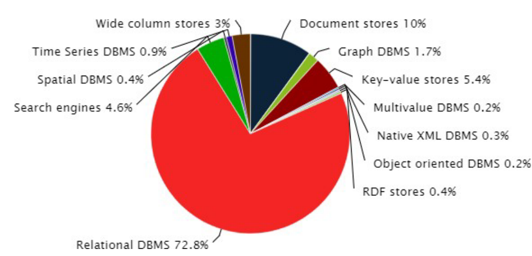

Chart 6: Ranking scores per category 2021

According to DB-Engines (2021a)'s monthly database popularity ranking as of Aug 2021, the top 10 databases are still dominated by traditional databases like Oracle, MySQL and Microsoft SQL, while NoSQL databases like MongoDB and Redis start sharing the market. This result echoes with Pokorny (2013)'s conclusion in his research who stated that NoSQL databases were still in development, and they would not replace relational databases till they became advanced database technologies.

373 systems in ranking, August 2021

Rank			DBMS	Database Model	Score		
Aug 2021	Jul 2021	Aug 2020			Aug 2021	Jul 2021	Aug 2020
1.	1.	1.	Oracle	Relational, Multi-model	1269.26	+6.59	-85.90
2.	2.	2.	MySQL	Relational, Multi-model	1238.22	+9.84	-23.36
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	973.35	-8.61	-102.53
4.	4.	4.	PostgreSQL	Relational, Multi-model	577.05	-0.10	+40.28
5.	5.	5.	MongoDB	Document, Multi-model	496.54	+0.38	+52.98
6.	6.	↑ 7.	Redis	Key-value, Multi-model	169.88	+1.58	+17.01
7.	7.	↓ 6.	IBM Db2	Relational, Multi-model	165.46	+0.31	+3.01
8.	8.	8.	Elasticsearch	Search engine, Multi-model	157.08	+1.32	+4.76
9.	9.	9.	SQLite	Relational	129.81	-0.39	+3.00
10.	↑ 11.	10.	Microsoft Access	Relational	114.84	+1.39	-5.02

Table 2: Database ranking in DB-engines as of Aug 2021

However, from the following trend chart, it is shown that MongoDB and PostgreSQL are catching up rapidly since 2013 and joined the other 3 major relational databases in top 5 popular databases.

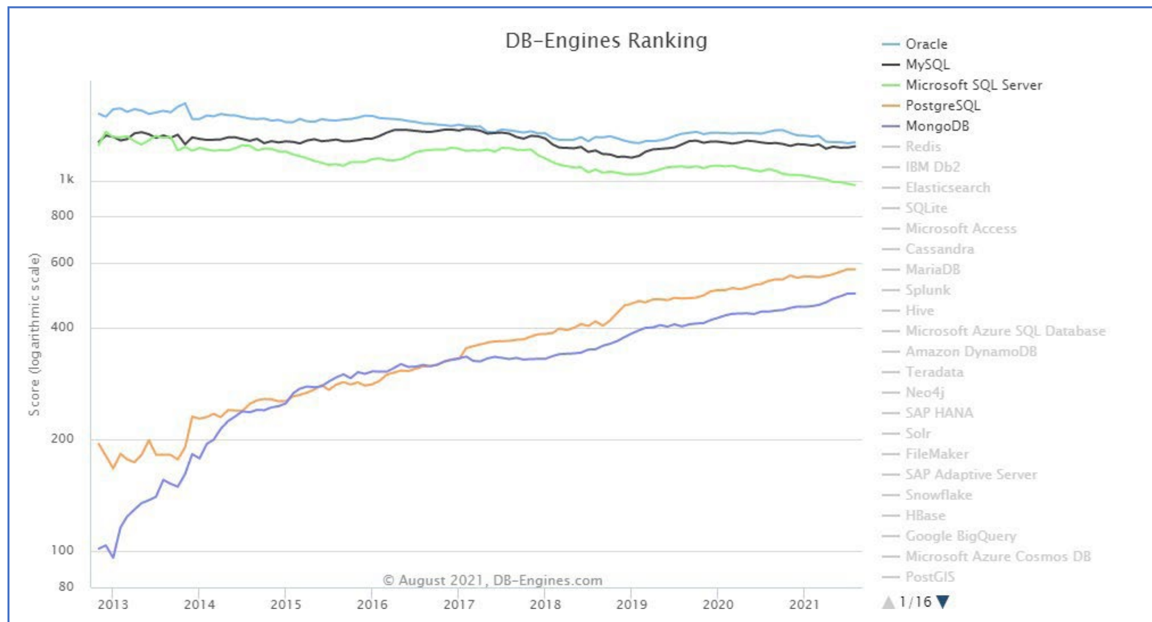

Chart 7: DB-Engines ranking – trend popularity (DB-Engines, 2021b)

Therefore, in this research, we will use MongoDB, the most popular NoSQL, and PostgreSQL the fastest growing SQL to represent each category and conduct the efficiency test. NewSQL databases will not be researched as most of them are relatively new, not ranked high in DB-Engine, and do not have many hits in IEEE Xplore database (Grolinger et al., 2013).

CAP Theorem

The CAP Theorem was brought up by computer scientist Eric Brewer at a symposium in 2000, so it is also known as Brewer's Theorem, which states that a distributed system can only simultaneously support two out of these three properties (De Angelis et al., 2018):

- Consistency

A guarantee that every user receives the most recent data.

- Availability

A guarantee that every request receives a response, either successful or failed

- Partition Tolerant

A guarantee that the system continues to work despite of message loss or communication cut down.

Based on this theorem, Singh (2018) mapped some relational and NoSQL databases to CAP like the following picture, from which, it is noticed that all NoSQL meet Partition-Tolerance while relational databases guarantee consistency. However, it is interesting to see NoSQLs like HBASE, MongoDB and Redis fall into CP area while general NoSQLs are well known as "sacrificing consistency for Availability and Performance". Singh (2018) explained further in his article that CP refers to systems which only sacrifice availability during network partition.

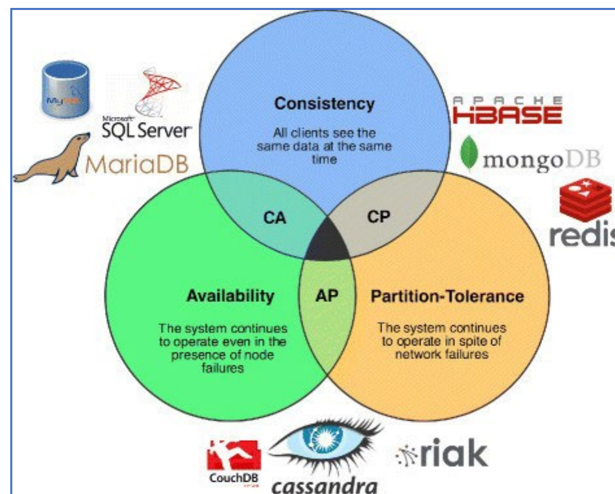

Figure 5: CAP Theorem and database management systems (Singh, 2018)

There were some arguments online discussing whether it is accurate to classify MongoDB under CP, as this database offers different consistency configuration. All reads in MongoDB go to the primary by default, but a developer can enable the function to read from secondaries to achieve eventual consistency. Therefore, JoCa argued on (Stack Overflow, n.d) that MongoDB actually is a trade-off among C, A and P, as its main focus changes according to the level of partition as shown in the following table. Based on this argument, people start realising that CAP may be an oversimple theory to define the complex scenarios in real world.

Scenario	Main Focus	Description
No partition	CA	The system is available and provides strong consistency
partition, majority connected	AP	Not synchronized writes from the old primary are ignored
partition, majority not connected	CP	only read access is provided to avoid separated and inconsistent systems

Table 3: Where does MongoDB stand in the CAP theorem? (Stack Overflow, n.d)

Database Selection

This section will review the advantages and disadvantages of using MongoDB and PostgreSQL, the two databases selected based on the above literature review findings.

MongoDB

MongoDB is an open-source document-store NoSQL database developed by a software company called 10gen (now called MongoDB Inc) using C++ in 2007 (Abramova & Bernardino, 2013). In MongoDB, data is stored as documents with unique ID and documents are serialised as JSON objects and stored as BSON (a binary encoding of JSON) (Makris et al., 2019). According to Abramova and Bernardino (2013), MongoDB uses a Master-Slave structure to ensure durability and concurrency, with one Master having write and read permissions, while Slave/Slaves functioning like a backup with only read permission. However, when Master is down, the Slave with the latest data will function as a Master. Even though Mongo is a NoSQL database, it has also adapted several relational database features, such as CRUD operations and indexing (Makris et al., 2019). Additionally, MongoDB can achieve strong consistency just like a relational database when there is no internet partition as it is a single-master system, and all reads go to the primary by default (Stack Overflow, n.d). As soon as the primary is down, a secondary will be determined as a new primary to achieve eventual consistency.

Being on the market for more than a decade, MongoDB also faces some criticisms, among which is the famous 2017 MongoDB security issue. In early 2017, thousands of MongoDB users were extorted by five groups of attackers for restoring deleted database data as their database was accessible to the public online using MongoDB default configuration (Computer World, 2017; Krebson Security, 2017). MongoDB later fixed this setting and started binding database to localhost instead. Other technical issues like roll back unacknowledged writes, reads and writes yield to record locks, and collation-based sorting, were addressed by MongoDB later (Wikipedia, n.d.-d).

MongoDB was chosen by the researcher to represent NoSQL, because it is free open source, and it is the most popular NoSQL database according to DB-engine ranking as of October 2021 (DB-Engines, 2021a).

PostgreSQL

PostgreSQL is an open-source object-relational database system (Makris et al., 2019), initiated by the Ingres project from University of California Berkeley (ACM, 2014) and called Ingres relational database systems in the 1980s (Stonebraker & Rowe, 1986). In 1996, two graduates from Berkeley replaced the POSTQUEL query language interpreter with the one for SQL and made it freely modifiable for the public (Postgres Help, 2019). In the same year, the database management system was renamed as PostgreSQL to show its support for SQL. PostgreSQL is a very powerful database and many researchers noticed that it even outperformed many commercial ones on the market. Therefore, people are curious on who are the group of people contributing to PostgreSQL. German (2006) noticed that there were only two people in the core team but a very large number of contributors who sent source code patches to the projects and report bugs, which is the true beauty of being open source.

When discussing PostgreSQL, many articles would mention its multi-version concurrency control, a mechanism to achieve data consistency and transaction isolation, also known as MVCC, where each transaction utilise a snapshot of the data which may be not up to date, to avoid changes to be made by other transactions (Nakamura et al., 2015). The problem of using MVCC is the cost of storing multiple versions of snapshots, and old versions are obsolete and need to be deleted to free up spaces (SAP Help Portal, n.d).

PostgreSQL was chosen by this research to represent relational database management system because of its outstanding performance in various evaluation tests of NoSQL and SQL (Makris et al., 2019; Treviño Villalobos, Víquez Acuña, & Quirós Oviedo, 2020), and the fact that it is free to be used for this research. Additionally, it is easy to be installed with lighter client library as it separates client and server (Jung, Youn, Bae, & Choi, 2015).

Relational databases

A relational database is a set of data where relationships are defined among them. Normally, in relational databases, data are presented in table format with columns and rows. Each column

stores a specific kind of data like string, integer or date, while each row collects a set of values of an object or entity with a unique primary key to be used to associate with rows in other tables as a foreign key (Amazon Web Services, n.d.).

Important concepts of relational databases

Apart from tables with columns and rows as the key concept of data storage, relational databases also have these four important aspects: SQL, Integrity, Transactions and ACID compliance.

SQL

Structured Query Language, also known as SQL, is a medium used to communicate with relational databases management systems in English-like statements, including insert, delete, update and query data (Amazon Web Services, n.d.). Like relational database, SQL was also invented by IBM in its System R project in early 1970s with the original name of SEQUEL (Structured English Query Language). It was renamed to SQL because SEQUEL was used as a hardware product (Sumathi & Esakkirajan, 2007, p. 112). In 1986, SQL became a standard of ANSI (American National Standards Institute) and supported by all types of relational database management systems.

There are three types of SQL commands: DDL, DML and DCL. DDL is short for Data Definition language commands to define a relational database and DDL includes creating, altering, and dropping tables and forming constraints. Data manipulation language commands (DML) are used to query or maintain the database, like inserting, updating, and querying data. DCL, data control language commands, is used to determine whether a user has the permission to conduct a specific operation (Sumathi & Esakkirajan, 2007, p. 114).

Data Integrity

In relational databases, data integrity covers three aspects: completeness, accuracy and consistency, which can be achieved by using different types of constraints: primary and foreign keys, and “unique”, “Not Null”, “Default” and “check” constraints (Amazon Web Services, n.d.).

These constraints are key elements of relational database as they enforce data accuracy and reliability and ensure business logic is followed.

Transactions

In relational databases, a transaction is a set of SQL statements executed as a series of operations, to form a unit of work. Transactions work on an "all or nothing" proposition, which means that the transaction must be completed successfully as a whole, otherwise data will be rolled back to previous status just like nothing happened. Therefore, the result of a transaction is either COMMIT or ROLLBACK. Each transaction is executed independently without impacting the result of other transactions, so multiple transactions can be executed concurrently. For example, if A paid $100 to B, the transaction would deduct $100 from A's account and add $100 to B's account. If there is any unexpected fault during the transaction execution, transaction will be rolled back and there is no change to either A or B's account.

ACID Compliance

ACID (Atomicity, Consistency, Isolation and Durability) is used to enforce data integrity.

Atomicity

Atomic enforces transactions to be executed as a whole and rolled back if unsuccessful, which is also known as "all or nothing" behaviour. This enforces the completeness of a business transaction which is crucial to many financial applications.

Consistency

Consistency makes sure that data written to the database comply to rules and restrictions defined including triggers, cascades and constraints (Amazon Web Services, n.d.). In other words, only data following all rules is allowed to be written to the database, otherwise, it will be rolled back to the previous state.

Isolation

Isolation ensures each transaction is executed independently and does not impact or is not impacted by other transactions. This property ensured huge number of transactions can be simultaneously processed independently on giant platforms like Amazon.

Durability

Durability means changes to the database are permanent after a transaction is completed successfully. Imagine a system outage happens on a running transaction, durability ensures transactions can be recovered fully and logs can be used to eventually write the failed transaction into the database permanently.

Many researchers recognised that ACID is the most important characteristic of relational databases, as it ensured valid database transactions (Aboutorabi[a], Rezapour, Moradi, & Ghadiri, 2015; Kunda & Phiri, 2017; Liu, 2015)

Advantages of Relational Databases

The largest advantage of relational database is its ACID compliance explained in the session above which enhances data consistency and security (Kunda & Phiri, 2017). Therefore, ACID compliance is particularly essential for industries:

- conducting monetary transactions
- handling time-sensitive data
- manage/monitor data in manufacturing, transportation and energy production (FairCom, 2018)

For financial systems, using an ACID-compliant database is vital as no customers want to encounter issues caused by inconsistent transaction processing like paying a fee twice, unable to access deposited money and conflicting updates. ACID compliance ensured data integrity in the most extreme scenarios like network outage, disruptions and equipment failures. Relational databases like MySQL, or Oracle can achieve ACID compliance because they use a locking

strategy where the database will lock the transaction that it is working on until the transaction completes successfully, which is similar to the scenarios of a user editing an excel sheet, nothing will be changed until the user clicks the save button.

ACID compliance distinguishes relational databases from other database formats like NoSQL database who have a built-in distributed systems architecture and cannot ensure complete transactional consistency. According to the CAP theorem mentioned in the previous chapters, distributed systems like NoSQL databases must choose one from full consistency and full availability but cannot achieve both. In most cases, NoSQL databases forfeited “Consistency” and “Isolation” while emphasised “Availability” and “Performance” and leave “consistency” to be handled by developers rather than databases (Chandra, 2015). Instead of complying with the ACID paradigm, NoSQL databases follow a new quasi-standard called BASE (cf. p. 33), a soft version of ACID and accept temporary data inconsistencies but enforces eventual consistent.

Additionally, relational databases have a standard query language (Kunda & Phiri, 2017), which made it easier and quicker to query the database especially when the data model is complicated with several tables involved and multiple relations built. According to GeeksforGeeks (2021), SQL can help retrieve large amount of data from database easily without using extra codes. It is a standard interactive language that can be easily learnt and understood to conduct complex queries portable to various devices and operation systems.

Disadvantage of Relational Databases

The most obvious disadvantage of using a relational database is its poor scalability to handle heavy website traffic and big volume of data (Kunda & Phiri, 2017). For relational databases, scalability is vertical which means more CPU, RAM, etc. are needed to be added to existing machines (Chandra, 2015). Technically, it is possible to scale a relational database horizontally, but it is very difficult because of hardware limitations, involvement of many complex Join operations and distributing data over many servers, and all of these require high-skilled technicians. Padhy, Patra, and Satapathy (2011) also pointed that the main problem of relational

databases is its limitation to scale with Data Warehousing, Grid, Web 2.0 and Cloud applications.

Relational databases with a strict schema make it difficult to alter the data structure and store data in different formats. Unstructured data has to be converted into relational structures in order to be saved in relational databases (Jung et al., 2015). Taking blogs as an example, relational database with strict schema is a heavy burden to this type of websites, when text, pictures and comments would be stored in separate tables, and it is made impossible to alter a blog's feature without system outage (Moniruzzaman & Hossain, 2013). Habib (2015) also indicated how difficult it is to add data into a predefined schema, as you need to migrate data from the existing one into a new schema which takes time and normally would cause system outage, and this may damage business reputation with customers.

Normalisation is a key step in designing a relational database management system, and Kroenke (1977) described this process is for "converting a relation that has certain problems to two or more relations that do not have these problems". Even the "father" of relational database Codd (1989) even admitted that "relational database is best suited to data with a rather regular or homogeneous structure", so it is common for DBAs to encounter problems when normalising heterogeneous data like images, text, and miscellaneous facts (Homan & Kovacs, 2009).

NoSQL

Background

NoSQL stands for "not only SQL" which does not store data in rows-and-column format, but in a more dynamic schema which is hard to visualise (Habib, 2015). Instead, NoSQL utilises different frameworks to organise data and is widely used in big data applications. According to Kunda and Phiri (2017), the development of web 2.0, 3.0 and big data has made NoSQL databases in high demand since 2000. Around 2009, developers gathered to discuss possibilities of an open-source, non-SQL database to store complex datasets.

The concept of petabyte datasets applied in Big Data Analytics, Business Intelligence, and social networking has pushed traditional database management systems to their limits (Moniruzzaman & Hossain, 2013). These limitations boosted the development of NoSQL options to achieve easy horizontal scalability and distributed data storage, e.g., Google's Bigtable and HBases (Chang et al., 2008), and Facebook's Cassandra (Lakshman & Malik, 2010). According to Chandra (2015), the key motivations of NoSQL development are easier development, growing data scale, having scalability and failover issues and can be used as a caching layer for transaction data storage.

Types of NoSQL databases

Chandra (2015) classified NoSQL databases into two categories: Aggregate Oriented and Non-aggregate Oriented, simply because Aggregate Oriented database "splits relations" while they both are Schema-Less. According to Chandra (2015), the former category re-arranges data collected from various nodes into different aggregate forms using MapReduce, which makes it useful if the same aggregate is used regularly. Many popular NoSQLs like key-value databases (Amazon S3, Voldemort and Scalarise), column databases (Cassandra and Hbase) and document-based databases (MongoDB) fall into this category. On the other hand, graph databases like Neo4J are typical non-aggregate oriented databases, which rely heavily on relations and is more ACID compliant.

However, NoSQL databases are more commonly categorised by their types as shown as the following table.

Type	Examples	Attributes
Column-Oriented Databases	Hbase Cassandra Hypertable	Data tables are stored by column rather than by rows, to reduce the data volume to be read from disk and to make it easy to compress data (Harizopoulos, Abadi, & Boncz, 2009). They are also seen as a hybrid between relational databases and NoSQLs.
Document-Based Databases	MongoDB CouchDB	Store and query data as JSON-like documents to optimise performance and availability (AWS, n.d.-a; Chandra, 2015).
Key-Value Stores	Tokyo Cabinet Tyrant Berkeley DB Memcache DB Rdis Voldemort	A key value is used to store data. These databases are highly partitionable and can be scaled horizontally (AWS, n.d.-b).
Graph Databases	InfoGrid Neo4J	A graph structure (nodes and edges) is used to represent and store data (Yoon, Kim, & Kim, 2017). It is more ACID compliant, and is suitable for applications like location-based services, recommendation engines and network-based platforms.
Object Database	Db4o Versant	Information is stored in the form of objects (Wikipedia, n.d.-b).
XML Database	Berkeley DB XML BaseX	Data is stored in XML format. It is a type of document database.

Table 4: Types of NoSQL database (Habib, 2015)

From the table above, it is noticed that there are too many types of NoSQL with too many different providers, which makes it even harder for developers to decide which one to use for their application. To make this decision process a little easier, Chandra (2015) designed a matrix based on database type and features as shown below. From this matrix, it is noticed that the Scalability and Functionality of Document Store databases are both marked “variable”, with Scalability assigned “Variable (high)” and Functionality assigned “Variable (Low)”, which may be because many document databases implemented various storage and access processing method including but not limited to BigTable, MapReduce and even columnar.

Data model	Performance	Scalability	Flexibility	Complexity	Functionality
Key-Value Stores	High	High	High	Low	Variable (None)
Column Store	High	High	Moderate	Low	Minimal
Document Store	High	Variable (high)	High	Low	Variable (Low)
Graph Database	Variable	Variable	High	High	Defined by Graph theory

Table 5: NoSQL database features (Chandra, 2015)

BASE vs ACID

Proposed by eBay and adopted by Amazon (Chandra, 2015), the BASE concept was deliberately invented to parallel the ACID paradigm, the core of relational databases (Chandra, 2015). BA in BASE is an acronym for Basically Available, where Available refers to the CAP Theorem as "the system continues to operate even in the presence of node failures" (Singh, 2018). S stands for "Soft State" with Soft meaning data may be inconsistent, and State means the systems state may change. E in BASE represents "Eventually consistent" meaning eventually accesses to the given data item will return the latest value (Vogels, 2009).

As shown in the following table, the three attributes of BASE: Basically Available, Soft state and Eventual Consistency match with Atomicity, Consistency and Isolation in ACID respectively. In contrast to the "all or nothing" principle and strict data consistency in ACID, BASE accepts temporary data inconsistencies but achieves consistency in the long run and relies on the application to guarantee consistency (Soft State). In other words, BASE prioritises Availability over Consistency. The main reason behind this philosophy is the major challenge that BASE faces — to update data distributed in multiple servers with unpredicted routes while data synchronization is extremely challenging (Chandra, 2015).

BASE	ACID
Basically Available	Atomicity
Soft State	Consistency
Eventual consistency	Isolation
	Durable

Table 6: BASE VS ACID

Chandra (2015) summarised the differences between ACID and BASE paradigms in details as presented in the following table, which clearly explained that BASE sacrifices Consistency and Isolation to achieve better Availability and Performance. This approach is extremely beneficial to systems that do not require strict consistency like financial systems do but face the challenge of system down caused by rapid growing data volume. Therefore, social media applications like Facebook and Twitter are among the best fit for BASE methodologies as it is acceptable to read an old Facebook post made a few minutes ago. However, just as Chandra (2015) summarised in the table, ACID builds a robust database requiring simple application code to work with, while BASE builds a simple database but requires much more complex code to update the distributed data.

ACID [C+A]	BASE [A+P]
Strong consistency	Accomplishes Consistency, Atomicity and Partition tolerance "eventually"
Isolation	Availability first
Focus on "commit"	Best effort
Nested transactions	Approximate answers
Pessimistic: Force consistency at the end of transaction	Optimistic: Accepts temporary database inconsistencies, Eventually Consistent
Difficult evolution	Simpler, Faster, Easier evolution
Suitable for Financial Portals	Suitable for non-financial web-based applications
Safe	Fast
Shared Something (Disk, Memory)	Shared Nothing
Scale UP (limited)	Scale Out (Unlimited)
Simple Code, robust database	Complex code, simple database
Single Machine	A cluster
CA	AP/CA/CP, i.e., any 2 out of 3.
Scale Vertically	Scale Horizontally
SQL	Custom APIs
Full Indexes	Indexing is mostly on Keys

Table 7: Characteristics of ACID and BASE (Chandra, 2015)

Benefits of using NoSQL

Unlike relational databases which follow a strict schema to define every row or tuple, NoSQL databases follow a “Schema-less” approach also known as “dynamic schema” (Chandra, 2015), which made it easier to add data into NoSQL databases than into relational databases, as it is not necessary to stop and refine the schema when inserting to the database (Habib, 2015). Therefore, for early-stage projects when a schema has not been established, NoSQL databases are a more convenient option as it would not cause system down time when adding new data into the schema.

NoSQL offers a cheaper and more stable solution for growing data volumes. For many years, information technology experts have relied on so-called vertical scaling (e.g., purchasing expensive bigger servers) to resolve issues arising from bigger data volume, which in many times is not reliable. On the other hand, NoSQL databases normally have better scalability and can achieve many businesses’ needs of horizontal scaling by adding/removing servers dynamically, which is more effective and cheaper (Pokorny, 2013). NoSQL databases typically employ a partitioning pattern called Sharding, which largely increased the scaling limits of these databases, where records can be partitioned into shards with an assigned partition ID, and these shards can be replicated if needed or split if the record is getting big (Chandra, 2015). NoSQL utilises its high scalability to easily spread big volume data storage and processing load to many different servers to improve performance, which is very economically efficient (Bhogal & Choksi, 2015).

NoSQL works better with cloud computing technology. In 2019, Gartner (a world leading research company among S&P 500) predicted that “the future of the database market is the cloud” and it also forecasted that by 2022, 75% of DBMSs will be migrated to the cloud, largely because the popularity of DMSA (databases management solutions for analytics) and SaaS (software as a service) model (Costello, 2019). This prediction was based on the fact that in 2018 cloud DBMSs contributed 68% of the total DBMS’s annual 18.4% growth to $46 billion.

The old model of centralised data storage has been largely revised by this new technology with

more applications utilising decentralised data stores because they are cheaper, more scalable, and more flexible (Hecht & Jablonski, 2011). This prevalent cloud technology requires the relevant database to have high dynamic scalability, which cannot be achieved by relational databases who commonly rely on one server and can only be scaled by adding extra processors, memory and storage which normally is very expensive (Pokorny, 2013). On the other hand, most NoSQL databases response is quicker as they use asynchronous replication which does not rely on bandwidth heavily and do not replicate simultaneously which makes it perform well when connection is not good (Pokorny, 2013). However, asynchronous replication has its risk of losing data as data is not replicated immediately and further locking is unavailable to protect data copies (Chandra, 2015).

NoSQL databases provide "a wider range of data models to choose from" (Nayak, Poriya, & Poojary, 2013). There are mainly four types of NoSQL databases grouped by data model: Key Value databases like Redis and DynamoDB; Column Oriented databases like Cassandra and Hbase; Document Databases like CouchDB and MongoDB; Graph databases like InfoGrid and Neo4J (Meysman, 2016). This variety of data models make it possible to use different NoSQL databases for specific areas of a system. For example, Facebook uses Cassandra for storage (Facebook Engineering, 2008), Hbase for messaging and monitoring (Mathew & Kumar, 2015), Neo4J for graphs (Neo4J, n.d.-a), and many other NoSQL databases for different purposes. Therefore, this approach may be a solution for ecommerce systems who may face the same issue of increasing numbers of users and products.

To sum up, NoSQL is in high demand for a business who may expand swiftly in future as it can scale horizontally and cost-efficiently. It works better with cloud computing and asynchronous replication technologies and can provide a wider range of data models to choose from for specific business areas.

Barriers to NoSQL adoption

The major barrier that has stopped many ecommerce systems from adopting NoSQL databases

is the lack of true ACID (Atomicity, Consistency, Isolation and Durability) transactions (Grolinger et al., 2013), which ensure data validity despite accidents like power off. Pokorny (2013) explained ACID properties as “all or nothing”, “the result of each transaction are tables with legal data”, “transactions are independent”, “database survives system failures”. For example, it is a single transaction to transfer money from one account to another, even when other changes like debiting one account is happening simultaneously (IBM Knowledge Center, 2020). In the real world, relational databases can achieve full ACID-compliance, while NoSQL cannot. For systems where accuracy is essential (e.g., banking systems), it may be easier to deal with performance problems than struggling with the lack of transactions support (Corbett et al., 2013). As technologies develop quickly, databases like CouchDB can achieve full ACID compliance (Chandra, 2015) while others like Cassandra have invented a tuneable consistency function where developers can decide to which degree the required data should be consistent, and this makes it possible for these kind of NoSQL databases to be used in real-time transaction processing (Pokorny, 2013).

Another concern that many developers have when adopting a NoSQL database is the lack of formalised query languages, mainly because NoSQL queries typically relate to physical level data models (Banerjee, Goto, Debnath, & Sarkar, 2017). Furthermore, Banerjee et al. (2017) also stated that this lack of standards affected the efficiency of query answering and made it harder for applications running on this type of database to be portable.

Lack of exact-match joins is another problem that many developers encountered with NoSQL databases. According to Kim and Shim (2015), NoSQL databases were designed to only support single-table queries that do not need joins, and joins in NoSQL databases are normally similarity joins instead of exact-match joins. However, this becomes an issue when applying these databases to a wide range of applications, which is very common in these days.

Last but not the least, it can be very expensive both in human (e.g., learning new skills or even employing new developers) and technical (e.g., change hardware) aspects to convert an existing relational database to a NoSQL database (Himango, 2017).

In summary, the main barrier that stops many businesses from using NoSQL is of the lack of ACID. Therefore, applications that require higher accuracy like banking systems are not early adapters of this technology, while social networking applications that do not rely heavily on real time accuracy are the first ones enjoying the “NoSQL cake”. However, other issues like lack of query standards and exact-match joins should not be disregarded.

Performance Testing

Benchmarks

Benchmarks play a very important role in evaluation of database performance and functionality, and they are important to IT professionals in determining IT investment and answer questions like how many users the system can hold concurrently, what the response times are under different workload and whether the system can be scaled easily (Jutla, Bodorik, & Wang, 1999). Some of these benchmarks even unconsciously played and continue to play a role to urge database vendors to improve their products and keep their databases competitive on the market. Wisconsin benchmark is a typical example of these kind of dominating benchmarks. As the first database performance test benchmark, Wisconsin was created by the Wisconsin Computer Sciences Department in the 1980s to evaluate the speedup characteristics of DIRECT (a database machine) (DeWitt, 1993). According to DeWitt (1993), this benchmark became so popular that database vendors constantly use the numbers to promote their products, while some vendors also complained that this benchmark did not represent their product fairly. However, Wisconsin was soon replaced by other benchmarks like Datamation, mainly because it is a single user benchmark, and it lacks bulk updates, which is essential in real-world applications. Based on Wisconsin, ANSI SQL STANDARD SCALABLE and PORTABLE (also known as AS3AP) was invented with a more complex workload including multi-user tests, unity functionality test, mixing batch and interactive queries (Gray, 1993; Makris et al., 2019; Zutshi, 1999). Additionally, the Transaction Processing Performance Council (TPC), a non-profitable organisation was funded to define benchmarks to evaluate transaction processing and database. They invented a series of benchmarks widely used by companies, researchers and academics, including TPC-A, TPC-B and TPC-C for evaluating relational database management systems and TPC-DI for evaluating data integration (Zhang, Lu, Xu, & Chen, 2019). However, the

above-mentioned benchmarks are mainly used for general database performance testing, while there are also specific ones created for evaluating ecommerce systems. However, various business models involved in ecommerce systems like B2B, B2C, C2C and C2B, make it impossible to use one benchmark to evaluate all types of ecommerce systems. TPC-W was invented to evaluate ecommerce systems, and it simulated user activities using a book shop website.

However, Menascé (2002), Garcia and Garcia (2003) summarised limitations of using TPC-W for ecommerce systems:

1. It is complex and time-consuming to use the software tool to process the benchmark.
2. It doesn't represent any specific application while ecommerce system requirements vary significantly among different business models.
3. It was only designed for workload generated by human beings, while many ecommerce systems support third party system requests whose workload can be much bigger.

However, the rapid development of NoSQL and big data boosted the emerging of the next generation of benchmarks, including YCSB, LDBC and BigBench. YCSB, short for Yahoo Cloud Serving Benchmark, is an open-source benchmark using Java Database Connector to connect with database systems with APIs to execute the performance test (Klostermeyer, 2021). YCSB is a two-tier (performance and scalability) multi-user benchmark (Ahmad Ghazal et al., 2013), and comes with six sets of workloads to evaluate NoSQL data stores: update heavily, read heavily, read only, read latest, short-range read and read-update-write (Ihde et al., 2021). Another widely used benchmark is BigBench, an end-to-end big data benchmark, which added semi-structured and unstructured data models to TPC-DS to evaluate databases and MapReduce systems with unstructured big data solutions (Ahmad Ghazal et al., 2013). BigBench was later adopted by TPC as TPCx-BB. However, Ghazal et al. (2017) pointed out a major limitation of the original BigBench which heavily relies on TPC-DC and treats web-logs as structured tables and processes queries using a fixed schema which is against the real-life scenarios. Therefore, Ghazal et al. (2017) proposed BigBench V2 with a semi-structured web-logs data structure and enhanced workload queries to address the shortcomings of the original BigBench and the later TPC-BB.

Benchmark challenges

TPC Enterprise Benchmark Standards

Year	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
TPC-A																																		
TPC-App																																		
TPC-B																																		
TPC-C																							10	5	4	2	1			1	3	4	3	
TPC-D																																		
TPC-DI																																		
TPC-DS																															1	5	4	3
TPC-E																							14	11	7	4	5	4	4	2	2	3	1	2
TPC-H																							10	20	5	9	16	5	10	5	4	9	4	30
TPC-R																																		
TPC-VMS																																		
TPC-W																																		
TPC Express Benchmark Standards																																		
TPCx-BB																													4	3	2	6	2	1
TPCx-HCI																																		
TPCx-HS																												15	8	1	6	14	4	5
TPCx-IoT																														1	1	5	2	12
TPCx-V																															1	2		1
Common Specifications																																		
Pricing																																		
Energy																																		
* ... active benchmark; ... obsolete benchmark; Benchmarks published since 2010 as of 06/30/2021																							34	36	16	15	22	24	26	13	20	48	20	54

Figure 6: TPC Benchmarks Overview
(Green background represents active benchmark, while brown background represents obsolete benchmark)

The above figure demonstrated the status of all TPC benchmarks and from where, it is noted that many of these benchmarks were terminated, including the famous one created specifically for evaluating ecommerce systems – TPC-W, while there were several express benchmarks which eventuated to make the process easier for users. The invention of a benchmark normally takes a long time, too long to be accepted by this rapid changing technology world. Additionally, it is difficult to have a benchmark to include cloud and data analytics which are increasingly important to an ecommerce System. Nambiar and Poess (2013) in summary analysed the downfall of TPC into three reasons:

1. Less income from the declining number of members caused by vendor mergers.
2. Manipulated benchmarks using unrealistic configurations.
3. Complexity to configure and run TPC benchmarks.

It is difficult to find a benchmark that can fit both SQL and NoSQL characteristics to conduct a performance test. Additionally, it is very time consuming to configure and setup tools and an environment compliant to a specific benchmark. Plus, most benchmarks are not free. Therefore, big scope research projects like Makris et al. (2019) decided to abandon the idea of using a benchmark to conduct performance tests between SQL and NoSQL after reviewing many popular ones on market, and instead used real world scenario queries.

JMeter

JMeter is an open-source java application to measure database performance using JDBC configuration (Rautmare & Bhalerao, 2016). Among many performance test tools like LoadRunner, WebLOAD, LoadUI and NeoLoad, JMeter is the most preferred tool and has been widely used by recent researchers to evaluate and compare database performance (Keshavarz, 2021; Rautmare & Bhalerao, 2016; Zaman et al., 2021) because it is free of cost and user friendly and most importantly it can be used to conduct performance tests for both SQL and NoSQL databases. Researchers can easily create a test plan and configure elements using JMeter and can access its comprehensive documentation to learn how to install and configure the tool. Keshavarz (2021) chose JMeter to compare MySQL and MongoDB because it calculates server response time without the extra work of creating test data and cleaning up and it is easy to scale the load by the number of threads or users. Nevedrov (2006) also recommended JMeter for web service performance tests as it is widely accepted in the IT industry and is highly extensible using APIs provided. Therefore, this research will use this free, powerful and widely recognised tool to conduct the performance test.

Similar research

In Liu (2015)'s research, he compared MySQL and Cloudant, a document-oriented database provided by IBM, for an ecommerce system called Soosokan. In his experiment, he designed four aspects to compare: Scalability, Time Complexity, Space and Economic Cost. For the scalability test, he used JMeter and tested these two databases' performance (response time, error rate and throughput) by increasing the number of users at a scale of 500 till 7000. Unfortunately, no major difference was found in this scalability test. However, in today's ecommerce era, it is common to see systems having millions of users e.g., Alibaba and Amazon. Therefore, it may be worth testing scalability at a larger scale for example ten thousand or even a million.

On the other hand, in the time complexity test (run "Query" and "Insert" with different number of entries till 10,000), Cloudant performed much better than MySQL, the tested relational database, but MySQL used space more efficiently than Cloudant. Regarding cost, Liu (2015) did a survey and found CloudAnt is much cheaper than MySQL.

On the other hand, Aboutorabi[a] et al. (2015) compared a group of SQL and NoSQL databases using CRUD operations (Create, Read, Update and Delete) in both a single database instance and a distributed environment. In this research, they ran each test 100 times instead of once, and used mean performance time to conduct comparison. The overall finding of their research was that NoSQL database performed better than traditional databases while MongoDB was the fastest when fetching data and CouchDB performed well for insert, update and delete operations.
However, they also noticed that SQL performed better with aggregate queries. Aggregate queries actually are very important to ecommerce systems as customers need to know how many search results they get after searching a keyword.

Unlike most researchers who conducted CRUD operations in one table, Keshavarz (2021) utilised the concept of two related tables, and executed each operation under two scenarios (one table and two related tables) in MySQL and MongoDB. Furthermore, the researcher increased the test scale to one million. In this research, Keshavarz (2021) found that MongoDB outperformed MySQL in all four CRUD operations and is easier for services that may require frequent changes in future and MongoDB does not require data to be structured.

CRUD operations are commonly used in database performance tests. Apart from the above-mentioned research conducted by Aboutorabi[a] et al. (2015), Truica, Boicea, and Trifan (2013) also used CRUD operations to compare MongoDB and MySQL, and González-Aparicio, Younas, Tuya, and Casado (2016) took the similar approach and tested NoSQL databases' CRUD operations and their consistency issues. Therefore, it is recognised that using CRUD operations methodology is a good start for comparing SQL and NoSQL database performance.

To make the performance comparison more accurate, Ohyver, Moniaga, Sungkawa, Subagyo, and Chandra (2019) used Wilcoxon Signed-Rank test to compare the performance of Firebase Realtime Database (NoSQL) and MySQL. Wilcoxon Signed-Rank test is a good strategy to compare test result data when there is no specific database performing dominantly better. However, in most research mentioned above, the test result is noticeable, and the researcher could easily decide which database performed better.

Unlike most above-mentioned research who used single thread tests, Flores et al. (2018) used multiple threads testing methodology where 99 concurrent requests were submitted to both SQL and NoSQL to evaluate the performance. Additionally, they tested MongoDB in both Windows and Linux and noticed that SQL server performed better than MongoDB in Windows environment, but MongoDB on Linux performed better than SQL on Windows. It is biased as they only tested MongoDB on Linux but did not conduct the same test for SQL, so it is unfair to conclude that MongoDB performed better than SQL in their research.

Regarding the times of each CRUD operation run in their tests, Ohyver et al. (2019) ran 100 times for most queries and used mean score to compare the result, while Aboutorabi[a] et al. (2015) ran 50 times and also used average response time as the final result to compare. However, Li and Manoharan (2013) used the average time over five runs, as they noticed that the absolute time values are not major. Therefore, the author of this research decided to run each CRUD query 10 times and add more runs if results vary largely. For the number of operations, Li and Manoharan (2013) used 10, 50, 100, 1,000, 10,000 and 100,000, Liu (2015) started from 500 and added 500 each time till 10,000 and Truica et al. (2013) used 1,000, 10,000 and 100,000. Therefore, it is reasonable and common to use 100, 1,000, 10,000 and 100,000 to represent different size of requests. On the other hand, for large scope of ecommerce systems, it is common to have concurrent requests submitted to databases especially during peak seasons like double 11 festival in Taobao and the prime day in Amazon. Therefore, the author decided to include size of 60, 120, 240, 480 and 960 concurrent requests in this research to test database scalability.

Performance tests on MongoDB vs PostgreSQL

Makris et al. (2019) conducted a performance test between MongoDB and PostgreSQL for spatio-temporal data. Unlike most researchers who normally use one node for testing, Makris et al. (2019) compared these two databases in both 1 node and 5-node cluster setup scenarios. In their research, it was found that PostgreSQL outperformed MongoDB in all queries with a more noticeable difference in the 5-node cluster setup. They also noticed that indexing had improved the query response time of PostgreSQL more significantly than MongoDB.

The research result from Jung et al. (2015) is contrary to the one of Makris et al. (2019). In their research, they executed CRUD operations with 30,000, 90,000, 150,000, 210,000 and 300,000 data cases in both PostgreSQL and MongoDB in a single node test environment. However, they configured two MongoDB data models: one is similar to the relational model; the other is totally unstructured. It is noticed that unstructured MongoDB outperformed PostgreSQL largely in all four operations and only the structured MongoDB lost the select operation to PostgreSQL slightly. Therefore, they concluded that MongoDB with an unstructured data model would suit systems with big unstructured data.

There may be several reasons why these two research projects ended with reverse results. The main reason would be the type of data they used for each project, with Jung et al. (2015) having used card milage data and Makris et al. (2019) tested spatio-temporal data. Another reason would be Makris et al. (2019) used PostgreSQL with the PostGIS extension which may be the main reason why PostgreSQL performed better in their research. For research method, Jung et al. (2015) used the traditional CRUD queries, while Makris et al. (2019) used a set of mimic real-world queries in the spatio-temporal field.

On the other hand, Schmid, Galicz, and Reinhardt (2015) conducted several performance tests with PostgreSQL and MongoDB for Web Map Service (WMS) using JMeter. WMS is a user interface where clients can get maps with visualised geospatial data and gain detailed information for specific features on the map. In their experiment, they utilised JMeter to simulate different users to query both PostgreSQL and MongoDB. In their cases, concurrent

requests with a normally small-time offset are very common in a production environment, so they performed simultaneous request tests with different time offsets via JMeter. In their research it is noticed that NoSQL databases like MongoDB do not have enough geo-functionalities, while SQL databases is far superior on these functionalities. It is a pity that Neo4J was not selected in their research to represent NoSQL, as it has powerful map functions (Neo4j, n.d.-b). Instead, their research found that MongoDB outperformed PostgreSQL on queries purely on attribute information, while PostgreSQL performed better on more complex geometry queries.

From these three performance tests on MongoDB and PostgreSQL conducted by Makris et al. (2019), Makris et al. (2019) and Schmid et al. (2015), it is observed that there is no simple answer like MongoDB is faster than PostgreSQL or PostgreSQL is better than MongoDB. It really depends on the type of data these performance tests were conducted with and the degree of complexity of test queries that researchers used. Therefore, JohnGPL commented on a blog written by Mohammed (2015b) that if the data is standalone and document like, then MongoDB would be a better option than PostgreSQL, but in most real-life cases, data is more related than standalone, so it is not surprising to see PostgreSQL outperformed MongoDB in many performance tests. Additionally, as a document-like database is relatively new to many developers who are so used to configuring relational databases, they tend to configure MongoDB like a relational database which would heavily impact its performance later.

Literature review summary

Due to the rapid development of ecommerce especially since Covid-19 lockdowns, many ecommerce systems especially B2C systems are facing scalability and performance issues with their traditional SQL configurations. Replacing the existing SQL with NoSQL becomes a critical decision that many businesses need to make. After reviewing the database market and multiple database performance research studies, the author decided to use PostgreSQL to represent SQL and MongoDB to represent NoSQL, with traditional CRUD operations integrated on JMeter. To define different sizes of ecommerce systems, the author decided to use a single node system and sizes of 100, 1,000, 10,000,

and 100,000 for single thread tests and sizes of 60, 120, 240, 480 and 960 for multiple threads tests.

Research Questions

The purpose of this research is to compare the performance of NoSQL and SQL for different sizes of ecommerce systems, and to help ecommerce platform developers make better database decisions to avoid future performance issues.

Main research question

Is NoSQL database more efficient than SQL database for different sizes of ecommerce systems? To answer this question, this research decided use MongoDB to represent NoSQL and PostgreSQL to represent SQL, on a single node. To define different sizes of ecommerce systems, this research decided to use sizes of 100, 1,000, 10,000 and 100,000 records to conduct the single thread test, and use 60, 120, 240, 480 and 960 concurrent users to conduct the multiple thread test.

Sub-questions

1. Will NoSQL perform better than SQL in CRUD operations on small ecommerce platforms?
2. Will NoSQL perform better than SQL in CRUD operations on big ecommerce platforms?
3. Will NoSQL perform better than SQL in scalability testing?

Hypotheses

To answer the above asked questions, these hypotheses are listed to forecast the result.

1. NoSQL will outperform SQL on small ecommerce platforms.
2. NoSQL will outperform SQL on big ecommerce platforms.
3. NoSQL will outperform SQL on high concurrency situation.

Research Design

To test the performance of NoSQL and SQL databases, the processing time of executing CRUD (create, read, update, and delete) operations using both types of databases will be used and compared. Therefore, this experiment procedure will be divided into five parts: selecting databases, designing database schema, importing ecommerce data, designing use cases and implementing experiments.

Selecting Databases

In this experiment, MongoDB and PostgreSQL are chosen to represent NoSQL and traditional relational database separately, as MongoDB is ranked number one in NoSQL databases by DB-engine, and PostgreSQL is the fastest growing SQL in these years (DB-Engines, 2021b). Another important reason to choose these two database systems is that they are both open-source and can be used for free in this experiment. The following table shows the version details of these two databases, and they are the latest version at the time of conducting this experiment:

Database	**Version**
MongoDB	v4.4
PostgreSQL	v12

Table 8: Versions of selected databases

Designing Database Schema

The schema of a real-world ecommerce database tends to be extremely complicated mainly because of the complex components structure covering supplier, shopping cart, catalogue, order, payment, shipping, and other components. The following chart presented the components structure of a typical ecommerce system.

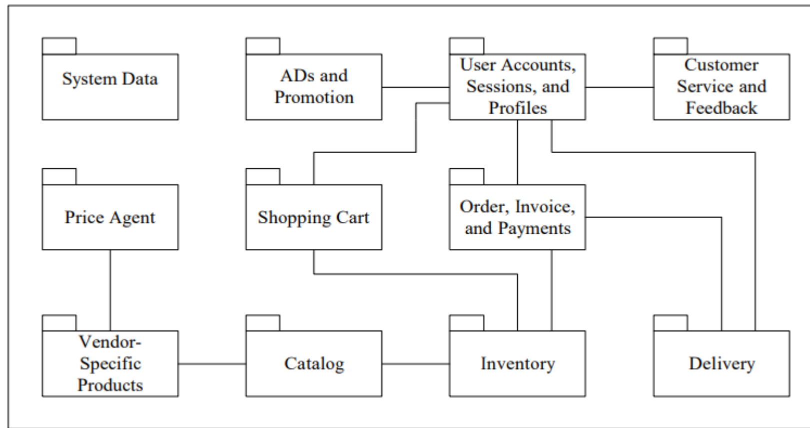

Chart 8: The components of a typical ecommerce system (Song & Whang, 2000)

It is difficult to find publicly accessible commercial database schemas and datasets online for the above complicated and similar ecommerce systems. However, there is another live ecommerce website called Brazilian Olist store which has made their database schemas public. Moreover, its real commercial datasets which cover the period from 2006 to 2008 are also published to Kaggle website (Kaggle, n.d.-a). Although the data related to companies and partners has been anonymised and replaced with the names of Game of Thrones great houses, it does not impact the experiment process of comparing the efficiency between NoSQL and SQL. In order to understand the Olist store datasets better, figure 7 presented the data schema provided by Olist, the largest department store in Brazilian marketplaces. Olist connects small businesses across Brazil to channels with ease, so that those traders can offer their products through the Olist store and ship products directly to customers using Olist specified logistic partners.

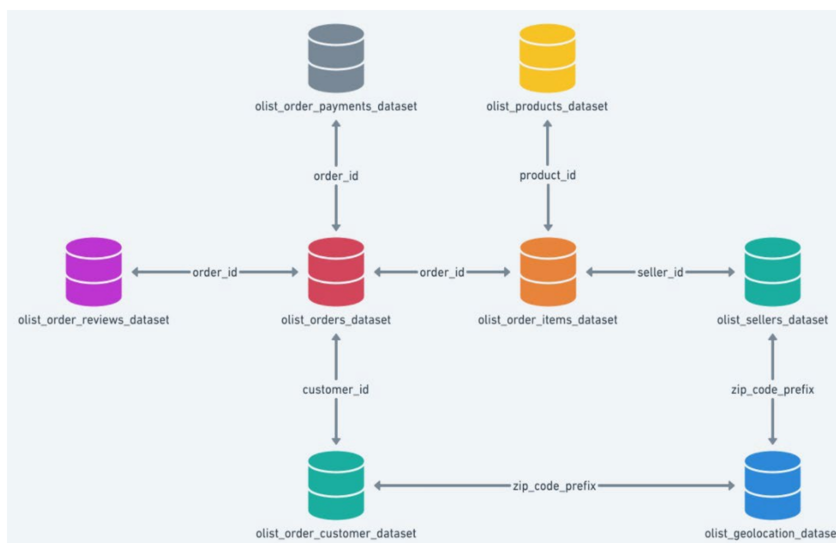

Figure 7: The data schema of Brazilian Olist store (Kaggle, n.d.-a)

The customers dataset contains information about customers and their locations, which aims to identify the unique customer in the orders dataset and to find the delivery location of the specific order. Each order in the orders table references to the primary key customer_id of the customers table, which means that the same customer can have multiple orders. Each order also contains specific order status and datetime when the order was generated and updated. When an order is shipped, the status will be updated to “shipped” and the datetime will be recorded. When an order is delivered, the status will be updated to “delivered” and the datetime will be documented. For the order items dataset, it includes information of items purchased within each order. For example, an order has three items which can be the same and each item has a freight fee calculated through its measures and weight. To get a total order value, an example calculation is shown as follows:

*The total order_item value is: 30.55 * 3 = 91.65*
*The total freight value is: 12.11 * 3 = 36.33*
The total order value is: 91.65 + 36.33 = 127.98

Besides, each order item references to the primary key seller_id of the sellers dataset and product_id of the products dataset. The sellers dataset includes data about the seller who fulfilled the order at Olist. It is used to find the seller location and to identify which seller completed each product. For the products dataset, it records data about the product, like category name it belongs to, product attributes and so on. The geolocation dataset is used to plot maps and calculate the distance between the seller and the customer, so it includes data of Brazilian zip codes and lat/lng coordinates. The payments dataset contains the orders payment information, like payment type, payment value, payment instalments and so on. For the order reviews dataset, it includes reviews information written by customers. After a customer purchases a product from Olist Store, the seller will be notified to complete the order. Once the customer receives the product, or the estimated delivery date is due, the customer gets a satisfaction survey via email to write a comment about the purchase experience.

In order to compare the performance efficiency between NoSQL (MongoDB) and SQL (PostgreSQL), five essential datasets will be extracted from the data schema of the Brazilian Olist store: olist_customers_dataset, olist_sellers_dataset, olist_products_dataset, olist_orders_dataset, and olist_order_items_dataset.

PostgreSQL

These five tables will be used and associated by foreign and primary keys to form the Entity Relationship Diagram (ERD) of the database as shown in figure 8, in which it indicates that the primary key customer_id of the olist_customers_dataset table is referenced by the olist_orders_dataset table that mainly contains order status and datetime. The olist_order_items_dataset contains order details and three foreign keys: order_id, product_id and seller_id, which refer to primary keys in orders, products and sellers tables. In addition, this diagram also presents one-to-many relationship between every two tables.

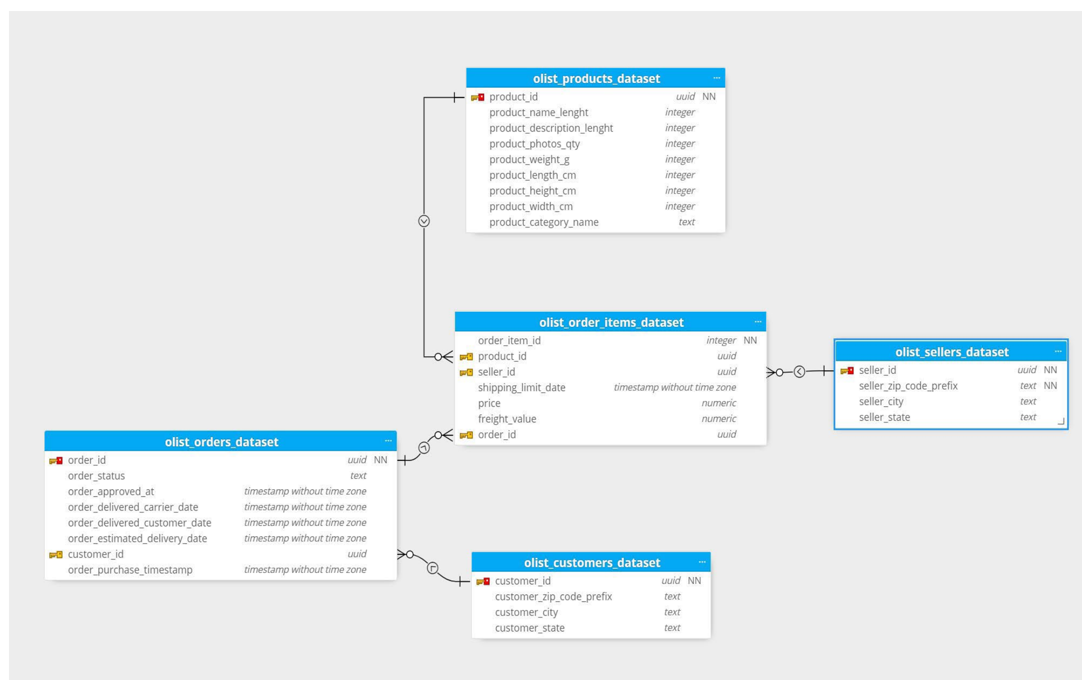

Figure 8: Entity relationship diagram for PostgreSQL

MongoDB

NoSQL databases like MongoDB do not support relationships between tables to form a fixed database schema. MongoDB stores data as collections in Binary Encoded JSON formats, and it is possible for a collection to contain various types of documents, as it has no pre-defined schema. A document can have a very complex structure containing array, or even nested documents (Győrödi, Győrödi, Pecherle, & Olah, 2015). In this experiment, four collections will be designed in the MongoDB database: olist_customers_dataset, olist_sellers_dataset, olist_products_dataset, and olist_orders_dataset. It does not have the fifth collections like what is configured in the PostgreSQL database because olist_order_items_dataset has been embedded in the

olist_orders_dataset collection in MongoDB, as it provides an embedded documents data model to describe a relationship between linked data without a necessary reference (MongoDB Documentation, NA-a, NA-b). Figure 13 shows the entity relationship diagram of the MongoDB database in which the order_items with a green caption is nested in the orders collection. Instead of using foreign keys to associate tables, MongoDB provides references to join data collections. As shown in figure 9-1, the orders collection with order_items has three references: customer_id, product_id and seller_id, which referring to primary keys respectively in customers, products and sellers collections. Therefore, in MongoDB, order information including order status and order details for a specific customer can be retrieved easily by this query:

db.olist_orders_dataset.find ({customer_id:ObjectId('60cefdc593fa145a40afd686')})

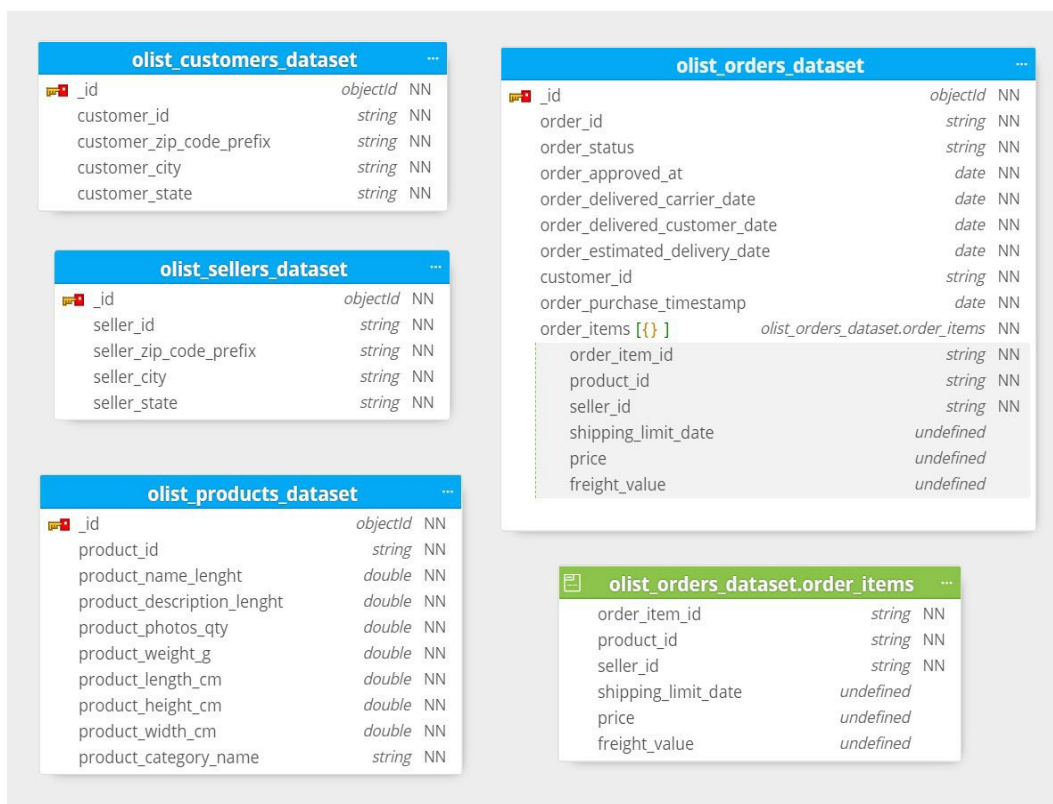

Figure 9-1: Entity relationship diagram for MongoDB

Importing Data

The author determined to use the above database schema because Olist represents a typical "Online intermediaries" B2C marketplace system who does not own products sold on the platform but instead provide a platform for buyers and sellers to connect with each other and based on the

literature review above, this type of platform would be the first facing scalability and performance issues comparing to other ecommerce types. Additionally, Olist provided its commercial data covering the period from 2006 to 2008 and published it to the Kaggle website where the data can be downloaded for free in csv format. Kaggle allows authors to use these data for non-commercial purposes and users can simply sign up for an account and agree to the terms of use (Kaggle, n.d.-b) to download a data sample. Regarding the data import process, the data in csv format is supported by both PostgreSQL and MongoDB. For PostgreSQL, the import operation is performed in windows command and the commands are shown below:

- **Import customers data**: copy brazilian_ecommerce.public.olist_customers_dataset (customer_id, customer_zip_code_prefix, customer_city, customer_state) from '/usr/local/archive/olist_customers_dataset.csv' csv header;
- **Import orders data**: copy brazilian_ecommerce.public.olist_orders_dataset (order_id, customer_id, order_status, order_purchase_timestamp, order_approved_at, order_delivered_carrier_date, order_delivered_customer_date, order_estimated_delivery_date) from '/usr/local/archive/olist_orders_dataset.csv' csv header;
- **Import products data**: copy brazilian_ecommerce.public.olist_products_dataset (product_id, product_category_name, product_name_lenght, product_description_lenght, product_photos_qty, product_weight_g, product_length_cm, product_height_cm, product_width_cm) from '/usr/local/archive/olist_products_dataset.csv' csv header;
- **Import sellers data**: copy brazilian_ecommerce.public.olist_sellers_dataset (seller_id, seller_zip_code_prefix, seller_city, seller_state) FROM '/usr/local/archive/olist_sellers_dataset.csv' CSV header;
- **Import order items data**: copy brazilian_ecommerce.public.olist_order_items_dataset (order_id, order_item_id, product_id, seller_id, shipping_limit_date, price, freight_value) from '/usr/local/archive/olist_order_items_dataset.csv' csv header;

For MongoDB, there are a serial of database tools which are a collection of command-line utilities to work with MongoDB deployment. One of the utilities is called “mongoimport” by which the

data can be imported into MongoDB. The commands are shown below:

- **Import customers data**: mongoimport --type csv -d brazilian_ecommerce -c olist_customers_dataset --headerline --drop '/usr/local/archive/olist_customers_dataset.csv'
- **Import products data**: mongoimport --type csv -d brazilian_ecommerce -c olist_products_dataset --headerline --drop '/usr/local/archive/olist_products_dataset.csv'
- **Import sellers data**: mongoimport --type csv -d brazilian_ecommerce -c olist_sellers_dataset --headerline --drop '/usr/local/archive/olist_sellers_dataset.csv'
- **Import orders data**: mongoimport --type csv -d brazilian_ecommerce -c olist_orders_dataset --headerline --drop '/usr/local/archive/olist_orders_dataset.csv'

The order items data cannot be imported directly into the order_items array object embedded in the orders collection due to the data format in the csv file which was designed for a relational database instead of a NoSQL database. In this case, an extra tool was developed to import this kind of data. These codes will be attached in the Appendix at the end.

Designing Use Cases

Normally, an ecommerce system contains various functions like searching, ordering, payment, and sometimes even review and online chat. It is well-known that a B2C ecommerce website is for businesses to sell commodities to consumers over the internet, and these types of ecommerce systems normally have some core components including searching for products, adding products into shopping cart to form an order, checking order status, and deleting order after it is completed. Based on these key functions of a B2C system, the following use cases are designed:

- Searching products
- Placing orders
- Updating order status
- Deleting historic orders

In order to compare the efficiency of MongoDB and PostgreSQL databases, these use cases would be simulated as if they are operated by real customers in an ecommerce website. Additionally, batches of 100, 1,000 and 10,000 records will be used to test these operations and observe how efficient these two databases will be when handling increasing data volumes.

Setup Experimental Environment

This section mainly describes the experimental process for both PostgreSQL and MongoDB based on above designed use cases. Before starting the experiment of performance comparison, a list of resources including hardware and software involved in the experiment environment will be listed below. Please note that the following listed hardware is home-grade instead of enterprise-grade due to a limited research budget. Therefore, future research can be done with enterprise-grade hardware setup, and multiple nodes.

Hardware Devices

There is only one computer involved in this experiment. In other words, both server-side and client-side share the same computer. In this way, the experimental results will not be affected by external factors like computer performance and WAN/LAN network, so the response time for comparing performance tends to be fair and accurate. The following table presents core specifications of the computer.

Computer	Server-side
CPU	Intel(R) Core (TM) i9-9900KF CPU @ 3.60GHz 64-bit
RAM	32 GB
Hard disk	1TB SSD
Operation system	Ubuntu 20.04 LTS

Table 9: The computer core specifications

The computer was equipped with the Ubuntu 20.04 LTS which is a Debian-based Linux operation system. Not only does the operation system have stable support for PostgreSQL and MongoDB databases, but also it is free.

Databases

As mentioned earlier, PostgreSQL and MongoDB were selected for this experiment to represent SQL and NoSQL respectively and test the performance efficiency of these two types of databases. Both database applications and configurations were established in a server-side with Ubuntu 20.04 LTS operation system. For PostgreSQL software, it is included in Ubuntu by default, so the installation can be easily executed by the apt-get command below:

apt-get install postgresql-12

APT (Advanced Package Tool) is a user-friendly command line tool to manage a packaging system that provides programs for installation. In this way, people can use it conveniently to locate and install new packages, upgrade existing packages and clean old packages (Ubuntu, n.d.). To install MongoDB software on the Ubuntu system, the official package called mongodb-org is involved, which is maintained and supported by MongoDB incorporation and the command to install the latest stable MongoDB (version 4.4) is as follows:

apt-get install -y mongodb-org

Performance Testing Tool

In this experiment, Apache JMeter will be used as a performance testing tool to conduct the efficiency comparison of PostgreSQL and MongoDB, as this tool supports different database types and has been widely used and recommended in many web services' performance test research studies including Nevedrov (2006) and Rautmare and Bhalerao (2016). Apache JMeter is an open-source application developed in the Java language. It was originally designed to test web-based applications but has since been developed to conduct other tests, such as performance testing and load testing. Moreover, JMeter can be utilised to test performance on both static and dynamic web applications. It can also be used to simulate a heavy load on a server or a group of servers to analyse overall performance (JMeter, n.d.).

Designing Experiments

This section will describe the experimental procedure of the performance and scalability tests on PostgreSQL and MongoDB using Apache JMeter. In JMeter, the PostgreSQL database is

connected using JDBC, a Java-based data access technology. The core settings include JDBC driver class object, database server, database name, username, and password. MongoDB database is connected to JMeter by the Apache Groovy script. This connection script will be attached in the appendix. Groovy script a powerful object-oriented programming language for the Java Platform, with both programming and scripting language features, like Python.

In these experiments, each performance test will be processed based on pre-designed use cases mentioned above. In addition, scalability tests will be included to measure these two databases' performance on scaling up or scaling out. The aim of the scalability test is to determine the number of concurrent users for each database and ensure user experiments in ecommerce systems.

To measure database performance, the response time of various requests on these two databases will be returned from each database to JMeter. In this process, the external response delays caused by the network and JMeter can be ignored as these databases and JMeter were all installed in the same device. Those experimental results will be presented in charts for better visualisation and easier comparison of performance between PostgreSQL and MongoDB.

Product Search

In this experiment, there will be three different database queries. The first one will be queries of searching products, the second one will be aggregate queries on products and the last one will be queries of searching a single product for scalability testing.

Search Queries

For an ecommerce website, the most important function for users is the product search function which is used to identify products users are looking for. Normally, there are two methods to locate products users want. The first is typing the keywords that match with relevant product attributes in the search box and these attributes can be product name, category, or brand, etc. The second is referring to search facets that have several different types, such as categories, product price, size and so on. Search facets can help users refine their options quickly, so they do not have to look for a specific product by browsing pages of information. In whichever way users seek products, the

technology behind these methods is the read operation by which the process is executed.

To compare performance of searching products between PostgreSQL and MongoDB, three database queries will be created and performed in JMeter which can be used to simulate the search function. The number of search results will be 100, 1,000 and 10,000 respectively. Each number represents the quantity of a specific product category and the three product categories selected will be "food", "music" and "bed_bath_table" which are real data provided by the ecommerce platform Olist.

The following is an example of the database query in PostgreSQL:

```
select * from olist_products_dataset where product_category_name = 'bed_bath_table'
```

The database query statement of MongoDB in JMeter is written in Apache Groovy script. The full script for searching products is attached in the Appendix and the core script is showed below:

```
mongoDB.getCollection("olist_products_dataset")
.find(eq("product_category_name", "bed_bath_table"));
```

Indexing is important to improve database performance when conducting searching queries. Without indexing, the database would have to scan the entire table to find all matching data, which is an inefficient retrieving method (PostgreSQL, n.d). According to the result of the performance comparison research conducted by Martins et al. (2021), the global performance of PostgreSQL can be improved by 91% with proper indexes. Additionally, Makris et al. (2019) identified a similar finding when evaluation MongoDB and PostgreSQL for spatio-temporal data and noticed that PostgreSQL executed queries significantly faster after using indexes.

In this experiment, the B-trees index type will be applied to both PostgreSQL and MongoDB queries as this index type is supported by both databases. For example, the field product_category_name from the table olist_products_dataset will be indexed in both databases, which means that data pages containing three types of levels (root, intermediate and leaf) will be created and connected as a doubly linked list as shown in figure 9-2, to ensure that these pages can be scanned sequentially in either forward or reverse directions.

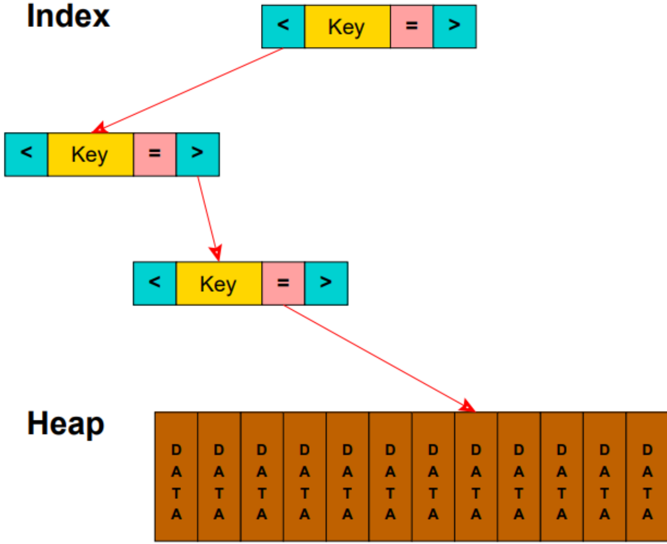

Figure 9-2: B-Tree index data structure (Momjian, 2021).

Aggregate Queries

Another important function for a database is the aggregate function used for calculations like calculating average value, counting number, summing values, finding the maximum value and the minimum value. As a result, there is an aggregate query which can be designed as another performance comparison between PostgreSQL and MongoDB. The term "aggregated query" is common in almost all database software documents. An aggregate query is a method of deriving group and subgroup data by analysing a set of data items. This query operation can also be thought of asking the database how to "group by". It may be helpful to think of the database as the "aggregate" and the information as the "query". In this experiment, "group by" function will be used to combine subsets by a specific category to calculate the number of products of the category. In this way, the number of products in each category will be calculated for three categories "food", "music" and "bed_bath_table" respectively.

The following is an example of the aggregate query in PostgreSQL:

```
select product_category_name, count(product_category_name) as number
from olist_products_dataset
where product_category_name='bed_bath_table' group by product_category_name
```

The aggregate query statement of MongoDB in JMeter is written in Apache Groovy script. The full script for search products is attached in Appendix and the core script is shown as below:

```
mongoDB.getCollection("olist_products_dataset").collection.aggregate(Arrays.asList(
Aggregates.match(Filters.eq("product_category_name", "bed_bath_table")),
Aggregates.group("$product_category_name", Accumulators.sum("count", 1))));
```

Scalability Test

The scalability test can be conducted easily using JMeter, as JMeter can not only simulate requests from one real user to the server, but also requests from multiple users. There are a couple of important settings in JMeter for the scalability test. The first one is the number of threads that will be used to simulate concurrent users to access the database. In this research, the number will be set to 60, 120, 240, 480 and 960 separately. The second one is the ramp-up period which is always set to 1 second. The shorter the period is set, the more accurate the scalability performance test. These two settings indicate that JMeter will take 1 second to respectively simulate 60, 120, 240, 480 and 960 concurrent users to perform the product searching function. Subsequently, processed results and the response time will be returned to JMeter from each database.

Order placement

In this experiment, there will be two different database insert operations. The first one will be operations of placing orders, and the second one will be operations of placing a single order for scalability testing.

Inserting Orders

In an ecommerce website, once users have identified the products they want, they would place orders on the order web page as the next step, while in the database, the process of placing an order is an insert operation. In this experiment, each order record references to the products table that contains product attributes like product name, quantity, and price. Each order also links to customer information such as shipping address. This experiment will involve three different scales

of order records, the sizes of which will be 100, 1,000 and 10,000 respectively. The purpose is to test the performance of insert operations in both PostgreSQL and MongoDB for different scales of ecommerce systems.

In PostgreSQL, there are two tables, olist_orders_dataset and olist_order_items_dataset, involved for the operation of placing orders. As mentioned earlier, olist_orders_dataset mainly stores summarised order information and customer information, whereas olist_order_items_dataset stores item details of each order. In order to test database insertion performance for different scales of ecommerce systems, three different batches (100, 1,000 and 10,000) of records will be inserted into one table - olist_order_items_dataset respectively. In this case, any order summary record will have to be inserted into the olist_orders_dataset table, which is to be referenced by the olist_order_items_dataset table. In JMeter, the insert statement of an order summary is shown as follows:

```
INSERT INTO public.olist_orders_dataset(
        order_id, order_status, order_approved_at, order_delivered_carrier_date,
        order_delivered_customer_date, order_estimated_delivery_date, customer_id,
        order_purchase_timestamp)
VALUES ('47770eb9-100c-2d0c-4494-6d9cf07ec65d','created',null,null,null,null,
        'b0830fb4-747a-6c6d-20de-a0b8c802d7ef','2021-08-31 13:00:00');
```

The following is the SQL statement of inserting 100 order details:

```
INSERT INTO public.olist_order_items_dataset
        (order_id,product_id, seller_id, shipping_limit_date, price, freight_value)
(SELECT '47770eb9-100c-2d0c-4494-6d9cf07ec65d',
        '4244733e-06e7-ecb4-970a-6e2683c13e61','48436dad-e18a-c8b2-bce0-89ec2a041202',
        '2021-09-01 13:00:00',58.90,13.29 FROM generate_series(1, 100) as x)
```

As MongoDB is a non-relational database, it does not natively support the model of relations. The olist_order_items_dataset table as a document is embedded into the olist_orders_dataset collection. In other words, each record in the olist_orders_dataset collection contains one order summary and one or more order details. In this experiment, one order summary respectively containing three different batches of order items (100, 1,000 and 10,000) will be inserted into the olist_orders_dataset collection. In JMeter, the full script of such an insert statement for placing orders is attached in Appendix and the core pseudocode script of placing 100 order details is

shown as follows:

```
List<Document> orderItems = new ArrayList<Document>();
Implementing loop statement (100 times) to assign values to the variable orderItems;
Document orderSummary = new Document();
orderSummary.append("order_items", orderItems);
mongoDB.getCollection("olist_orders_dataset").insertOne(orderSummary);
```

Scalability Test

Regarding the scalability test of inserting operation, the number of threads is set to 60, 120, 240, 480 and 960 separately, and the ramp-up period is set to 1 second. As a result, JMeter will take 1 second to simulate 60, 120, 240, 480 and 960 concurrent users respectively to place an order in the PostgreSQL and MongoDB environments.

Order update

In this experiment, there will be two different database update operations. The first one will be operations of updating orders, and the second one will be operations of updating a single order for scalability testing.

Updating Orders

The ecommerce platform Olist includes various order statuses after users have identified products and placed an order. In the Kaggle website, Olist also provided the real data for orders with statuses as created, approved, processing, invoiced, shipped, and delivered. These statuses could be changed from one to another throughout the entire purchase behaviour from generating a new order to completing the order. In addition, there are other two special statuses: cancelled and unavailable, which will be used when the order is cancelled or unavailable. The process of changing an order status is an update operation in a database system. This experiment is designed to update order records in different scales of ecommerce systems in both PostgreSQL and MongoDB.

For PostgreSQL, the order status field is stored in the olist_orders_dataset table which contains more than 90,000 records provided by the Olist platform. To change order status using SQL update statement, there are three different batches of records, 100, 1,000 and 10,000 records

selected from pre-defined customers in the experiment. The following script shows the SQL statement of updating 100 order records:

```
update public.olist_orders_dataset od
set order_status='approved', order_approved_at='2021-09-02 13:00:00'
from (select * from olist_orders_dataset where customer_id='06b8999e-2fba-1a1f-bc88-172c00ba8bc7') fo
where od.order_id = fo.order_id
```

When updating order status in MongoDB, the olist_orders_dataset collection storing order status field will be updated. In this experiment, there will also be three batches of records, 100, 1,000 and 10,000, selected based on the specific customer, and the order status of each record will be updated using Groovy script in JMeter. The following is the core script of updating 100 order records extracted from JMeter. The full script for updating order status is attached in Appendix.

```
mongoDB.getCollection("olist_orders_dataset")
.updateMany(eq("customer_id", "06b8999e-2fba-1a1f-bc88-172c00ba8bc7"),
combine(set("order_status","approved"), set("order_approved_at",
new Date())));
```

Scalability Test

Regarding the scalability test of the update operation, the settings in JMeter are the same with the inserting operation experiment, which will be 60, 120, 240, 480 and 960 threads respectively with one second ramp-up period. Consequently, JMeter will take one second to respectively simulate 60, 120, 240, 480 and 960 concurrent users to update orders in both PostgreSQL and MongoDB.

Order deletion

In this experiment, there will be two different groups of delete operations. The first one will be operations of deleting orders, and the second one will be operations of deleting a single order for scalability testing.

Deleting Orders

In an ecommerce website, the deletion behaviour is rarely triggered in user front-end interface, while it is processed on demand in the server side. When the number of purchases increases, the number of historic orders will also rise. In this case, there is a need to purge those old orders periodically in an ecommerce system. To compare the performance of executing delete operations

in PostgreSQL and MongoDB, there will be three different scales of historic orders, 100, 1,000, 10,000, to be deleted from the ecommerce system in this experiment.

For PostgreSQL, historic order records are stored in the olist_orders_dataset table, and those records to be deleted are selected based on a specific customer. The deletion operation will be executed using SQL statement in JMeter. The following script presents the SQL statement of deleting 100 order records as an example:

```
delete from olist_orders_dataset where customer_id="06b8999e-2fba-1a1f-bc88-172c00ba8bc7"
```

In MongoDB, historic order records are stored in the olist_orders_dataset collection which will be used to selecte and delete orders from a specific customer. In JMeter, a record is deleted from MongoDB using Groovy script. The following is the core script extracted from JMeter for deleting 100 historic order records, and the full script is attached in Appendix.

```
mongoDB.getCollection("olist_orders_dataset")
.deleteMany(eq("customer_id", "06b8999e-2fba-1a1f-bc88-172c00ba8bc7"));
```

Scalability Test

For the scalability testing of deleting operation, it has the same settings with other operation tests, which are 60, 120, 240, 480 and 960 threads respectively with one second ramp-up period. In this way, JMeter will take one second to simulate 60, 120, 240, 480 and 960 concurrent users to delete historic order records from PostgreSQL and MongoDB.

Experiment results and findings

This section will describe the results of all above experiments with pre-designed use cases. In these performance experiments, each operation of CRUD was run 10 times for 100, 1,000 and 10,000 records in JMeter. The response time of processing these operations was recorded in milliseconds. The author decided to perform each operation 10 times to minimise the possibility of any external elements that might impact the response time accidentally. For the four CRUD scalability tests, each user request was simulated in a multi-user environment of 60, 120, 240, 480 and 960 concurrent users. Each simulated request was run 10 times, and minimum, maximum, and

mean average response times were recorded in JMeter, but only the mean average response time was used for performance comparisons between the two databases.

These experiment results of each CRUD operation were analysed and compared separately in PostgreSQL and MongoDB. In addition, these results will be evaluated across all operations. Afterwards, a brief summary will be given at the end of this section.

Product search results and findings

Search Queries

Figure 10 presented all response times of ten samples for reading 100 records in each database. PostgreSQL had a mean average response time of 11.6ms across ten samples, while MongoDB had an average response time of 3.5ms, which was more than 2 times faster than PostgreSQL.

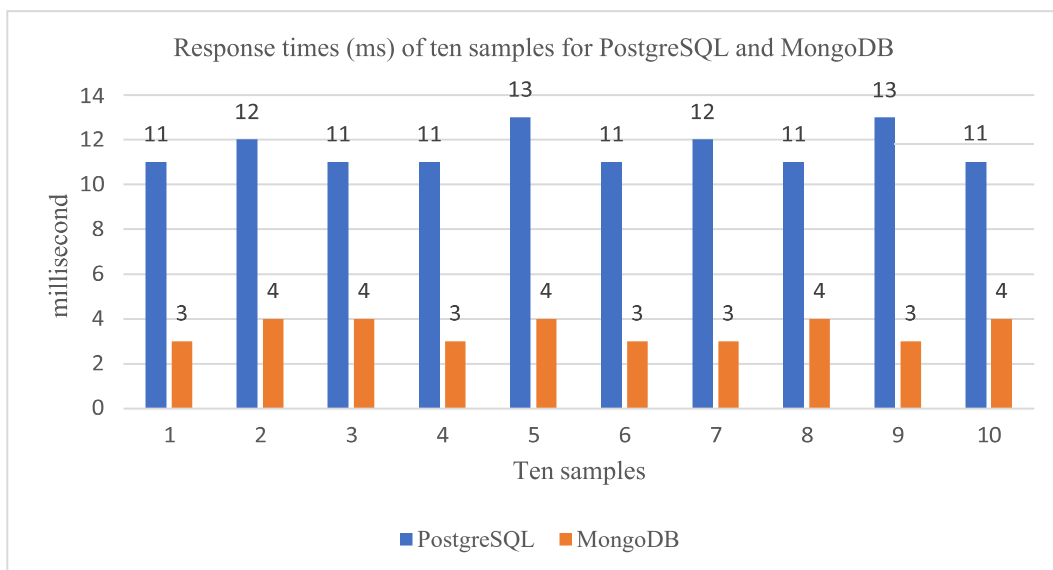

Figure 10: The experiment result of ten samples for searching 100 products in PostgreSQL and MongoDB

Figure 11 presented all response times of ten samples for reading 1,000 records in each database. PostgreSQL had an average response time of 12.2ms across ten samples, while MongoDB had an average response time of 4.8ms, which was more than 1.5 times faster than PostgreSQL.

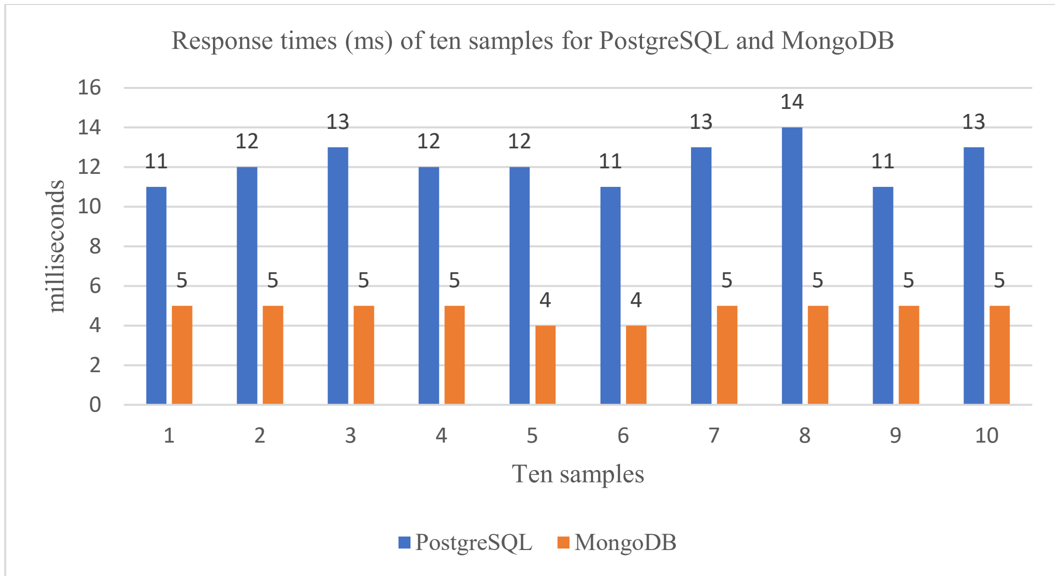

Figure 11: The experiment result of ten samples for searching 1,000 products in PostgreSQL and MongoDB

Figure 12 presented all response times of ten samples for reading 10,000 records in each database. PostgreSQL had an average response time of 35.4ms across ten samples, while MongoDB had an average response time of 23.9ms, which was 11.5ms faster than PostgreSQL.

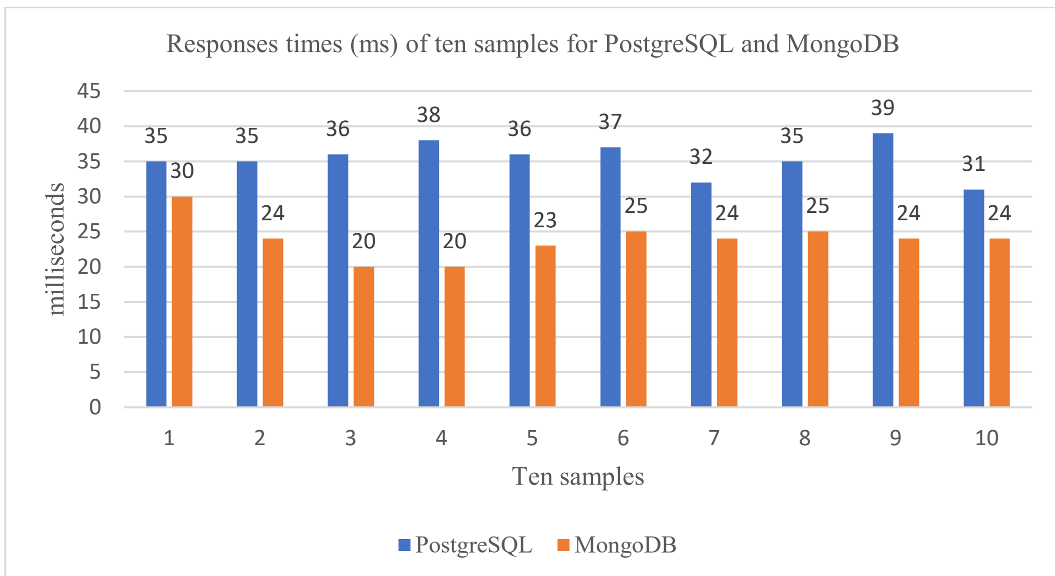

Figure 12: The experiment result of ten samples for searching 10,000 products in PostgreSQL and MongoDB

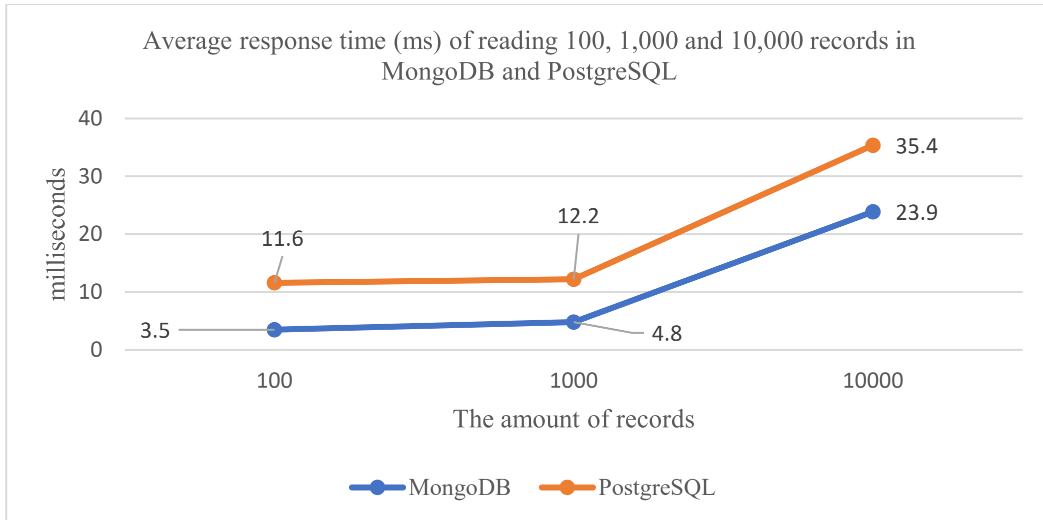

Figure 13: A comparison of average response time for reading different records (100, 1,000 and 10,000) between MongoDB and PostgreSQL

Regarding the searching function performance test, the results showed that the query performance was higher in MongoDB than in PostgreSQL with all average response times in MongoDB faster than in PostgreSQL when reading 100, 1,000 and 10,000 records. When reading 100 records, the average response time in MongoDB was 11.6ms, which was more than 3 times faster than PostgreSQL. The average response time of reading 1,000 records in MongoDB was 12.2ms, more than 2.5 times faster than PostgreSQL. When reading 10,000 records, the average response time in MongoDB was 23.9ms, comparing to the 35.4ms in PostgreSQL. This result revealed that search queries were executed almost 1.5 times faster in MongoDB than in PostgreSQL. Generally, it should take shorter time for the database to execute fewer records, while this test found that the average response time of executing 100 records were very close to the one of executing 1,000 records, with PostgreSQL took 11.6ms and 12.2ms respectively, and MongoDB took 3.5ms and 4.8ms respectively. This showed that it only took PostgreSQL extra 0.6ms to execute 900 search queries, while it took MongoDB extra 1.3ms, which indicated that PostgreSQL may outperform MongoDB when the number of records increase on a large scale. Additionally, there was a trend that these two average response times of reading 10,000 records in both databases were getting closer. To verify the hypothesis of PostgreSQL may outperform MongoDB when reading larger number of records, an additional test would be added for further comparison. In this additional test, 100,000 records would be inserted into both databases respectively so that these records can be retrieved by a specific query. The query for PostgreSQL was shown below:

*select * from olist_products_dataset where product_category_name ='toys'*

The core script for MongoDB was presented below:

List<Document> results = collection.find(eq("product_category_name", "toys")).toList()

The query was executed 10 times for each database, and the returned results were shown as follows:

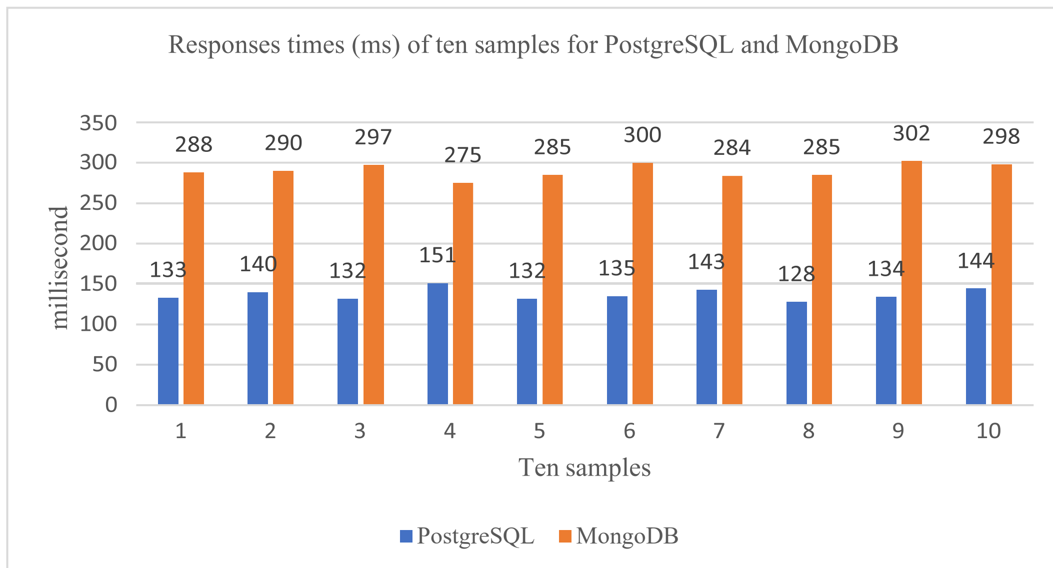

Figure 14: The experiment result of searching 100,000 products for PostgreSQL and MongoDB

Above figure 14 illustrated the response times of ten samples returned from each database after searching 100,000 records. In PostgreSQL, the average response time was 137.2ms over ten samples, while in MongoDB it was 290.4ms. Obviously, the average response time in PostgreSQL was much faster than MongoDB. Figure 15 showed that the average response time of each database was getting higher as the number of reading records increased. Surprisingly, PostgreSQL performed much better than MongoDB when the sample size increased to 100,000, while MongoDB outperformed PostgreSQL in sizes of 100, 1,000 and 10,000.

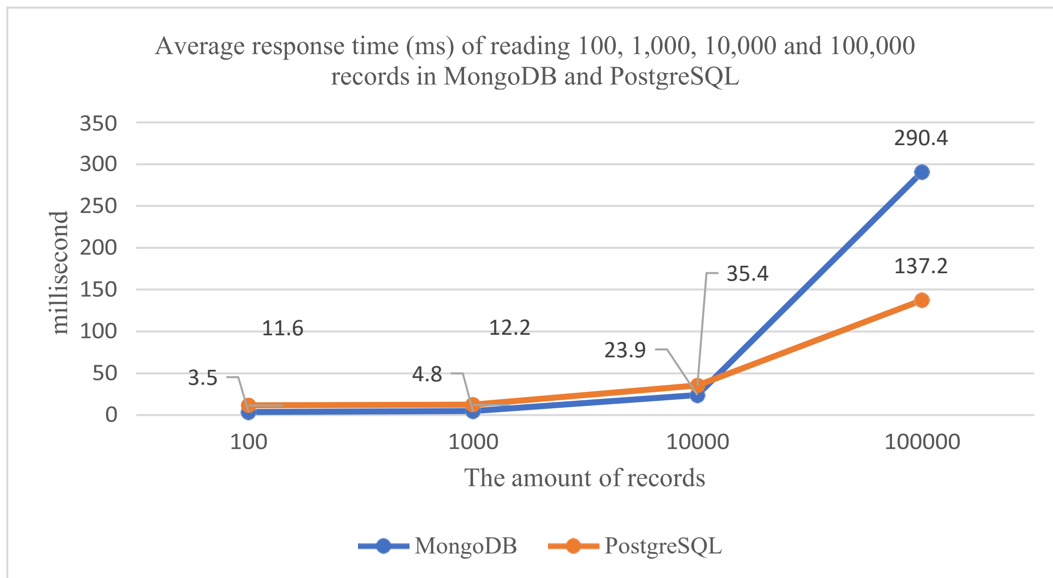

Figure 15: A comparison of average response time for reading different records (100, 1,000, 10,000 and 100,000) between MongoDB and PostgreSQL

In this experiment, queries were executed based on the same data model but different data schema. PostgreSQL is a table-based relational database having a pre-defined schema, while MongoDB is a non-relational database with dynamic schema storing unstructured data. These differences led to differences in data size and receiving speed. When reading 100,000 records, the returned data size in PostgreSQL was 7,682,476 bytes, while it was 12,133,465 bytes in MongoDB, which was almost twice the data size in PostgreSQL. Normally, the more records a database reads, the less time it spends on read each record. The following table showed the receiving speeds (KB/sec) when reading different numbers of records in PostgreSQL and MongoDB.

	100	1,000	10,000	100,000
PostgreSQL	117.24	1,563.67	12,037.17	40,055.62
MongoDB	3,656.08	12,529.7	32,631.07	33,738.86

Table 10: Receiving speeds (KB/sec) of reading different numbers of records in PostgreSQL and MongoDB

In MongoDB, the receiving speed was 33,738.86 KB/s for reading 100,000 records, and it was 32,631.07 KB/s for reading 10,000 records. Obviously, the receiving speed had barely improved. On the other hand, the receiving speed in PostgreSQL was 40,055.62 KB/sec when reading 100,000 records, 3.3 times of the speed when reading 10,000 records. As a result, the query

performance in PostgreSQL was more efficient than in MongoDB when reading a larger number of records, but for reading single record or a small number of records, MongoDB outperformed PostgreSQL. In ecommerce systems, reading 100,000 records/products at once is very rare, as developers normally would use pagination to limit the number of records can be returned at once (Yan, Cheung, Yang, & Lu, 2017). Therefore, in real life ecommerce systems, MongoDB tends to outperform PostgreSQL in retrieving products from a database more quickly.

Aggregated Queries

Figure 16 presented all response times of ten samples for the aggregate query of combining 100 records in each database. PostgreSQL had an average response time of 11.2ms across ten samples, while MongoDB had an average response time of 2.8ms, nearly 4 times faster than PostgreSQL.

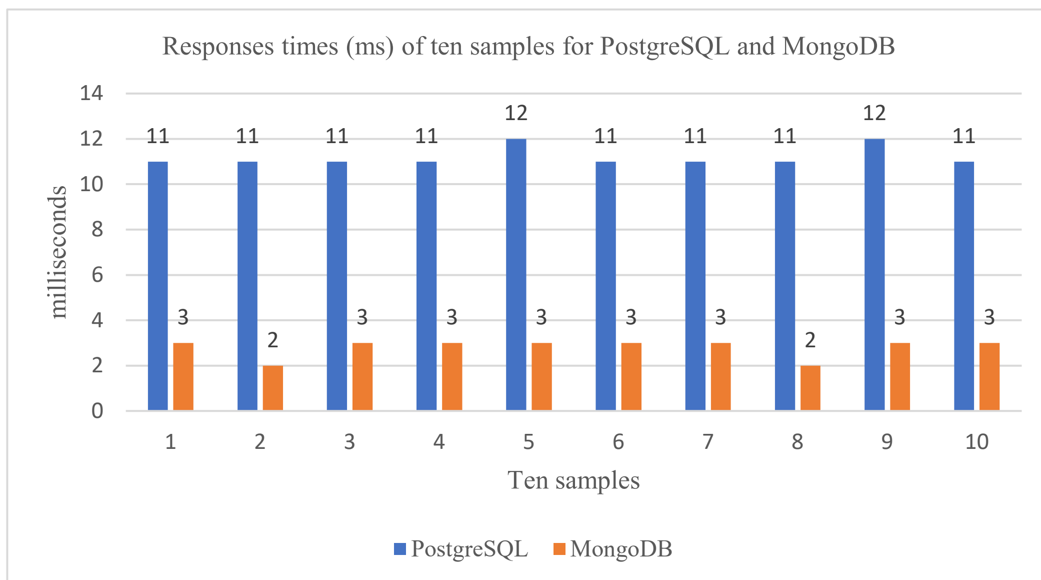

Figure 16: The experiment result of ten samples
for the aggregate query of combining 100 records in PostgreSQL and MongoDB

Figure 17 presented all response times of ten samples for the aggregate query of combining 1,000 records in each database. PostgreSQL had an average response time of 11.5ms across ten samples, while MongoDB had an average response time of 3.3ms, more than 2 times faster than PostgreSQL.

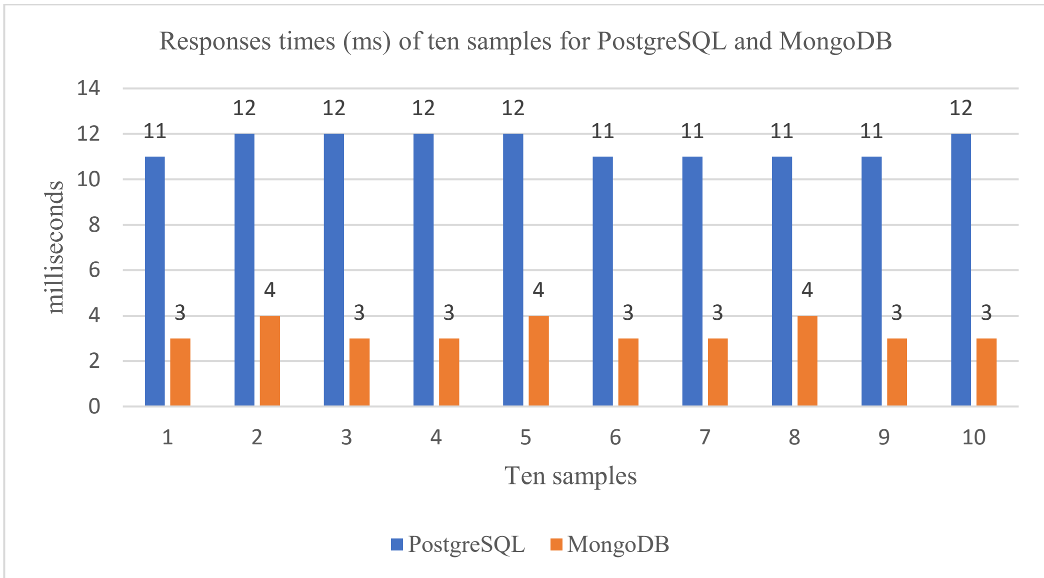

Figure 17: The experiment result of ten samples
for the aggregate query of combining 1,000 records in PostgreSQL and MongoDB

Figure 18 presented all response times of ten samples for the aggregate query of combining 10,000 records in each database. PostgreSQL had an average response time of 11.6ms across ten samples, while MongoDB had an average response time of 6.7ms, approximately twice faster than PostgreSQL.

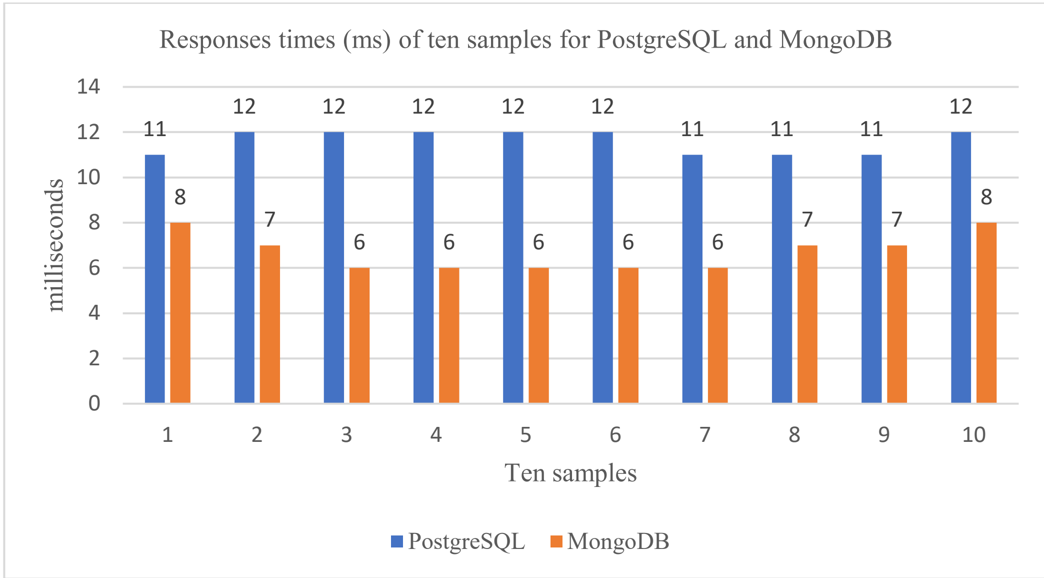

Figure 18: The experiment result of ten samples
for the aggregate query of combining 10,000 records in PostgreSQL and MongoDB

Figure 19 presented all response times of ten samples for the aggregate query of combining 100,000 records in each database. PostgreSQL had an average response time of 17.8ms across ten

samples, while MongoDB had an average response time of 30.7ms, approximately twice slower than PostgreSQL.

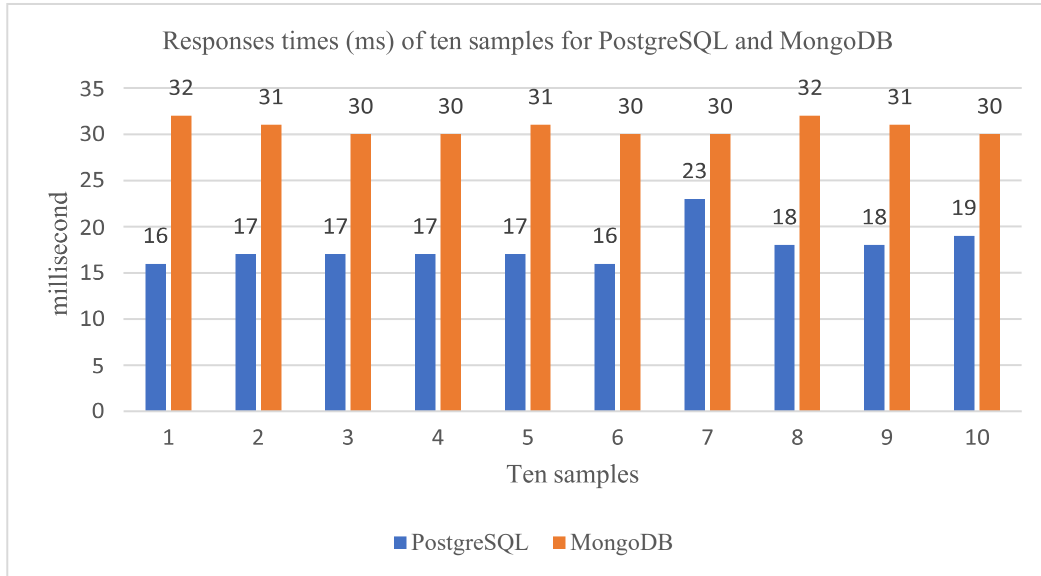

Figure 19: The experiment result of ten samples
for the aggregate query of combining 100,000 records in PostgreSQL and MongoDB

Regarding the performance of executing aggregate queries in both databases with batches of 100, 1,000, and 10,000 records, MongoDB was faster than PostgreSQL. For counting 100 records, the average response time in MongoDB was 2.8ms, 3 times faster than in PostgreSQL. The average response time for counting 1,000 records in MongoDB was 3.3ms, more than 2 times faster than PostgreSQL. When counting 10,000 records, the average response time in MongoDB was 6.7ms, while it was 11.6ms in PostgreSQL, which meant that the speed in MongoDB was nearly twice of the speed of PostgreSQL. Generally, the fewer records the database processed, the shorter time the database took, but the average response time for counting 100 records was very close to the average response time for counting 1,000 records, with PostgreSQL which ran for 11.2ms and 11.5ms respectively, and MongoDB ran for 2.8ms and 3.3ms respectively. Therefore, it only took PostgreSQL 0.3ms more to run extra 900 records, while it took MongoDB 0.5ms more. Additionally, the average response time was doubled when counting 10,000 records in MongoDB, while it was almost unchanged in PostgreSQL. As the number of records increased, the performance began to degrade, so when counting 100,000 records, the average response time in MongoDB increased from 6.7ms to 30.7ms, while it only increased by 6.2ms in PostgreSQL. In this case, PostgreSQL outperformed MongoDB in the scale of 100,000.

Figure 20 showed that the average response time of each database was getting bigger as the number of records increased, especially in MongoDB. The figure also presented performance comparisons for counting 100, 1,000, 10,000 and 100,000 records between MongoDB and PostgreSQL.

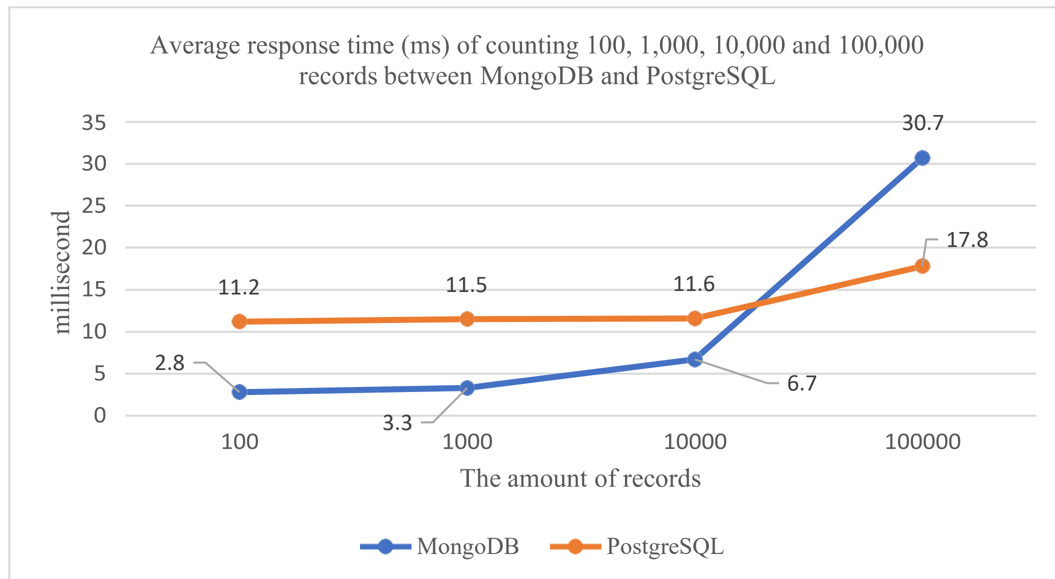

Figure 20: A comparison of average response time for counting different records (100, 1,000, 10,000 and 100,000) between MongoDB and PostgreSQL

As a result, the performance of the aggregate function in PostgreSQL was more efficient than in MongoDB when reading a large number of records, but for reading a single record or a small number of records, MongoDB outperformed PostgreSQL. In real ecommerce use cases, aggregated queries are used frequently to calculate the number of products returned after a search (Takanobu, Zhuang, Huang, Feng, Tang, & Zheng, 2019), and sometimes, there can be more than 100,000 products in the search result, so it is believed that PostgreSQL is a better option for large scale ecommerce systems.

Scalability Test

This research used average response time of each user's request in a simulated multi-user environment for scalability analysis comparisons between PostgreSQL and MongoDB. There were respectively 60, 120, 240, 480 and 960 concurrent users who searched a product in each database. Figure 21 illustrated that PostgreSQL got the average response times of 65ms, 70ms, 74ms, 81ms and 138ms, while they were 63ms, 111ms, 595ms, 1,819ms and 5,311ms in MongoDB.

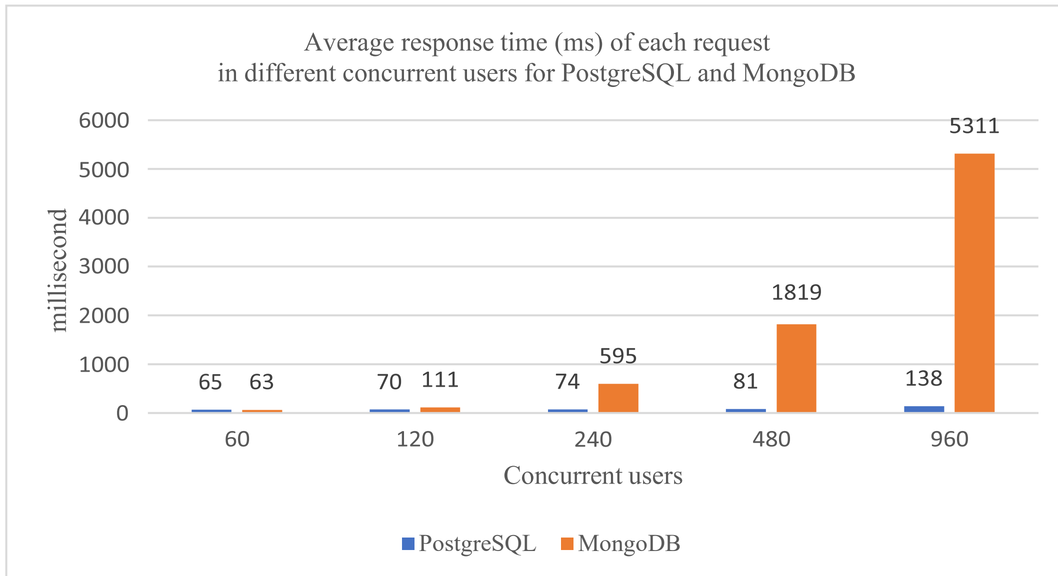

Figure 21: The experiment result of reading a record in a simulated environment with 60, 120, 240, 480 and 960 concurrent users for PostgreSQL and MongoDB

Regarding the scalability performance comparison of searching a product in both databases, the average response times of executing the queries in PostgreSQL were 65ms, 70ms, 74ms, 81ms and 138ms respectively for five different numbers of 60, 120, 240, 480 and 960 concurrent users, while they were 63ms, 111ms, 595ms, 1,819ms and 5,311ms respectively in MongoDB. This result showed that the average response time of each request in PostgreSQL was almost the same as MongoDB for the size of 60 concurrent users. For each request in 120 concurrent users, the average response time in PostgreSQL was almost twice as fast as MongoDB. In 240 concurrent users, the average response time in PostgreSQL was 8 times as fast as MongoDB. In 480 concurrent users, the average response time in PostgreSQL was 22 times as fast as MongoDB. This gap increased to 38 times in the size of 960 concurrent users. Obviously, the performance of each request in MongoDB degraded significantly as the number of concurrent users increased, while the performance in PostgreSQL was barely affected by the size of concurrent users, because the throughput of PostgreSQL almost doubled as the number of concurrent users increased. However, the throughput of MongoDB barely increased when the number of concurrent users increased to 120. Normally, database throughput is measured in requests per second for scalability testing (VMware, n.d.). Throughput is also related to the processing capacity of the underlying system, such as disk I/O, CPU speed, memory bandwidth and so on. As mentioned earlier, this experiment was conducted under the same software and hardware environment, so the only

parameter that could have affected the throughput was the performance of the specific database. Table 11 showed the comparison of throughput for different concurrency levels between PostgreSQL and MongoDB databases. As a result, PostgreSQL outperformed MongoDB when user requests were scaled up.

	60	120	240	480	960
PostgreSQL	59.8	126.1	250.5	501.2	996.9
MongoDB	58.3	119.9	127.8	127.2	123.7

Table 11: A comparison of throughput (requests/sec) for the five different numbers of concurrent users between PostgreSQL and MongoDB

Order placement results and findings

Insert Queries

Figure 22 presented all response times of ten samples for inserting 100 records into each database. PostgreSQL had an average response time of 11ms across ten samples, while MongoDB had an average response time of 5.1ms, twice as fast as PostgreSQL.

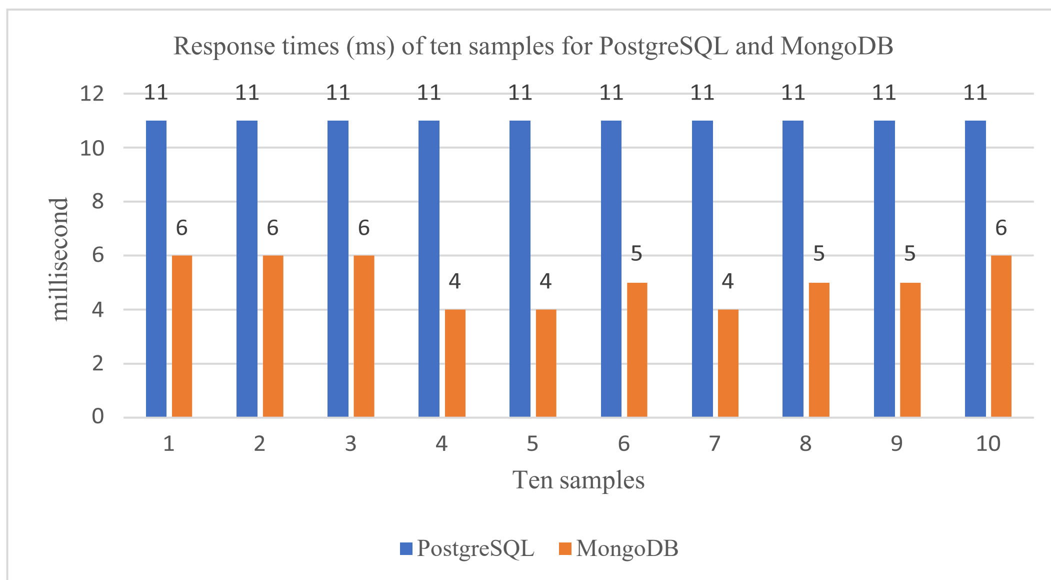

Figure 22: The experiment result of ten samples for placing 100 orders in PostgreSQL and MongoDB

Figure 23 presented all response times of ten samples for inserting 1,000 records into each database. PostgreSQL had an average response time of 104.6ms across ten samples, while MongoDB had an average response time of 6.6ms, 15 times as fast as PostgreSQL.

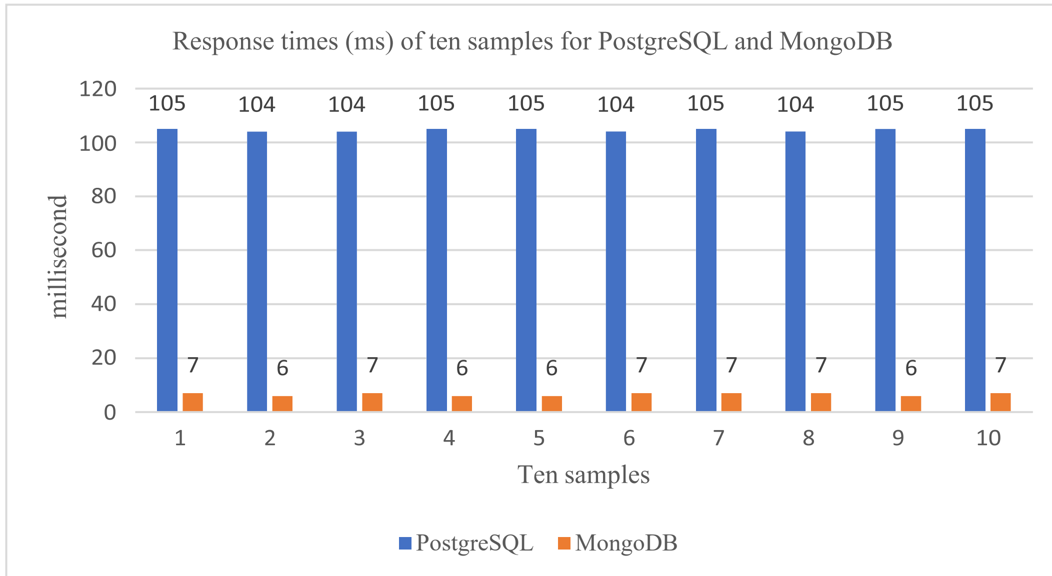

Figure 23: The experiment result of ten samples for placing 1,000 orders in PostgreSQL and MongoDB

Figure 24 presented all response times of ten samples for inserting 10,000 records in each database. PostgreSQL had an average response time of 1048.3ms across ten samples, while MongoDB had an average response time of 40.2ms, 26 times as fast as PostgreSQL.

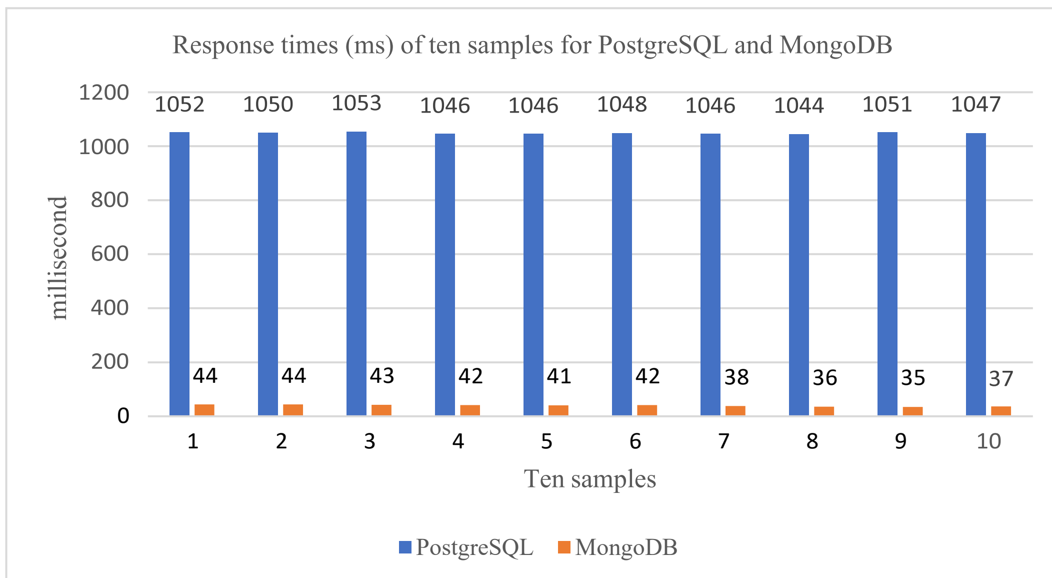

Figure 24: The experiment result of ten samples for placing 10,000 orders in PostgreSQL and MongoDB

Regarding the performance of placing orders, three average response times in MongoDB were much faster than the one in PostgreSQL when inserting 100, 1,000 and 10,000 records into database. The average response time in MongoDB was 5.1ms, while it was 11ms in PostgreSQL

for inserting 100 records. For inserting 1,000 records, the average response time in MongoDB was only 6.6ms, almost 16 times as fast as PostgreSQL. When inserting 10,000 records, the average response time in MongoDB was 40.2ms, while it reached at 1,048.3ms in PostgreSQL. This meant that the execution of the insert operation in PostgreSQL was more than 26 times slower. Generally, the fewer records the database inserted, the shorter time it took the database to insert. Figure 25 showed that the average response time of each database was getting longer as the number of records inserted increased, and it presented the performance comparison for inserting 100, 1,000 and 10,000 records into MongoDB and PostgreSQL.

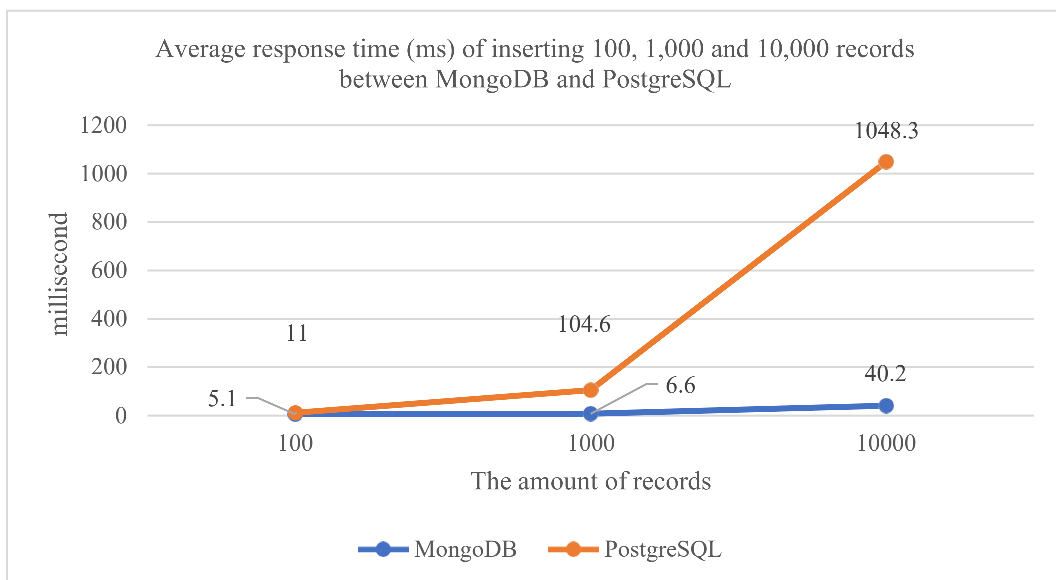

Figure 25: A comparison of average response time
for inserting different records (100, 1,000 and 10,000) between MongoDB and PostgreSQL

As the number of records increased from 100 to 10,000, the average response time increased by 35.1ms (from 5.1ms to 40.2ms) in MongoDB. In contrast, the average response time increased by 1037.3ms (from 11ms to 1048.3ms) in PostgreSQL. This meant that the increase rate of average response time in PostgreSQL was much bigger than in MongoDB for the same scale of growth. As a result, these aspects showed that MongoDB performed insert operations far more efficiently than PostgreSQL, especially when the number of records increase largely.

Scalability Test

This research used average response time of each user's request in a simulated multi-user environment for scalability analysis comparisons between PostgreSQL and MongoDB. There were

respectively 60, 120, 240, 480 and 960 concurrent users who placed an order in each database. The figure 26 illustrated that PostgreSQL got the average response time of 3ms, 4ms, 5ms, 5ms and 5ms, while they were 3ms, 4ms, 4ms, 5ms and 10ms in MongoDB.

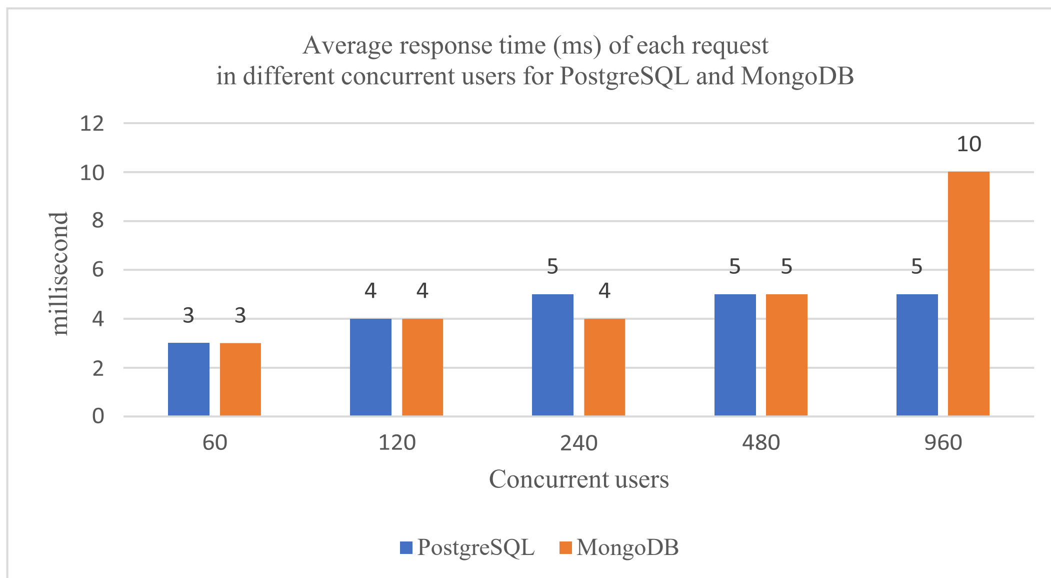

Figure 26: The experiment result of placing an order in a simulated environment with 60, 120, 240, 480 and 960 concurrent users for PostgreSQL and MongoDB

Regarding scalability performance comparison of placing an order, average response times of the insert operation in PostgreSQL were 3ms, 4ms, 5ms, 5ms and 5ms respectively for different numbers of concurrent users (60, 120, 240, 480 and 960), while they were 3ms, 4ms, 4ms, 5ms and 10ms respectively in MongoDB. The result showed that the average response time of each request in PostgreSQL was almost the same, while the performance of MongoDB had reduced as the number of concurrent users were increasing. For the performance of the insert operation in 960 concurrent requests, the average response time of each request in PostgreSQL was 5ms, twice as fast as MongoDB. However, for MongoDB, the average response time was 10ms with 960 concurrent requests, twice as slow as the average response time with 480 concurrent requests.

Obviously, as the number of concurrent users exceeded 960, MongoDB performance degraded significantly, while PostgreSQL performed stably when the number of concurrent users increased because the throughput of PostgreSQL was getting much higher than MongoDB when executing 960 concurrent requests. However, there was a very slight difference in throughput for sizes of 60,

120, 240, and 480 concurrent requests in these two databases. Table 12 showed the comparison of throughput for different concurrency levels between PostgreSQL and MongoDB databases. As a result, for this scalability testing, PostgreSQL performed better than MongoDB when user requests were scaled up.

	60	120	240	480	960
PostgreSQL	61.1	129.4	263.7	519.5	1051.5
MongoDB	59.9	124.6	251.6	498.4	950.3

Table 12: A comparison of throughput (requests/sec) for the five different numbers of concurrent users between PostgreSQL and MongoDB

Order update results and findings

Update Queries

Figure 27 presented all response times of ten samples for updating 100 records in each database. PostgreSQL had an average response time of 10ms across ten samples, while MongoDB had an average response time of 58.9ms, 5.8 times as slow as PostgreSQL.

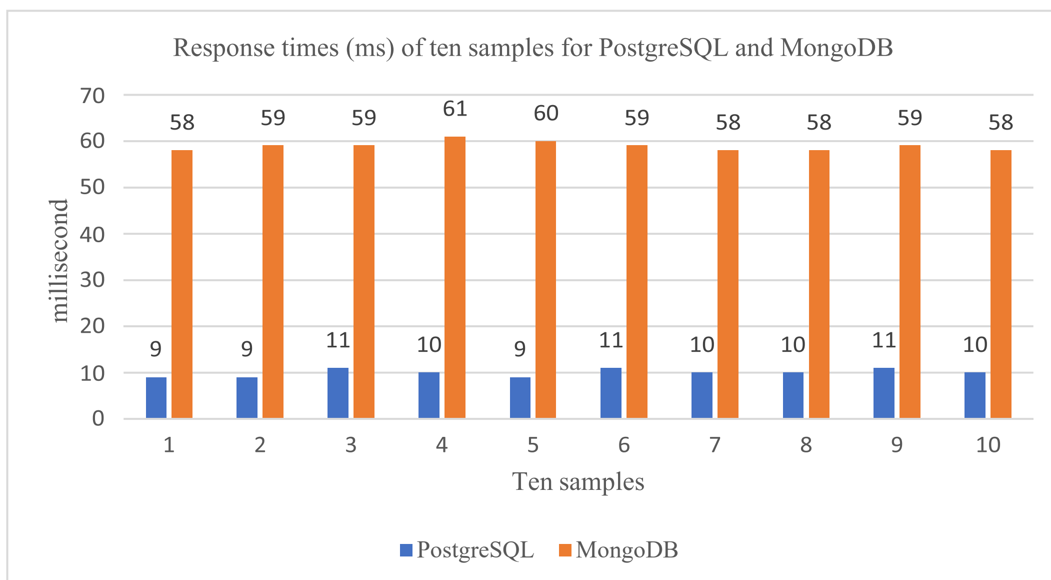

Figure 27: The experiment result of ten samples for updating 100 orders in PostgreSQL and MongoDB

Figure 28 presented all response times of ten samples for updating 1,000 records in each database. PostgreSQL had an average response time of 27.9ms across ten samples, while MongoDB had an average response time of 72.4ms, 2.5 times as slow as PostgreSQL.

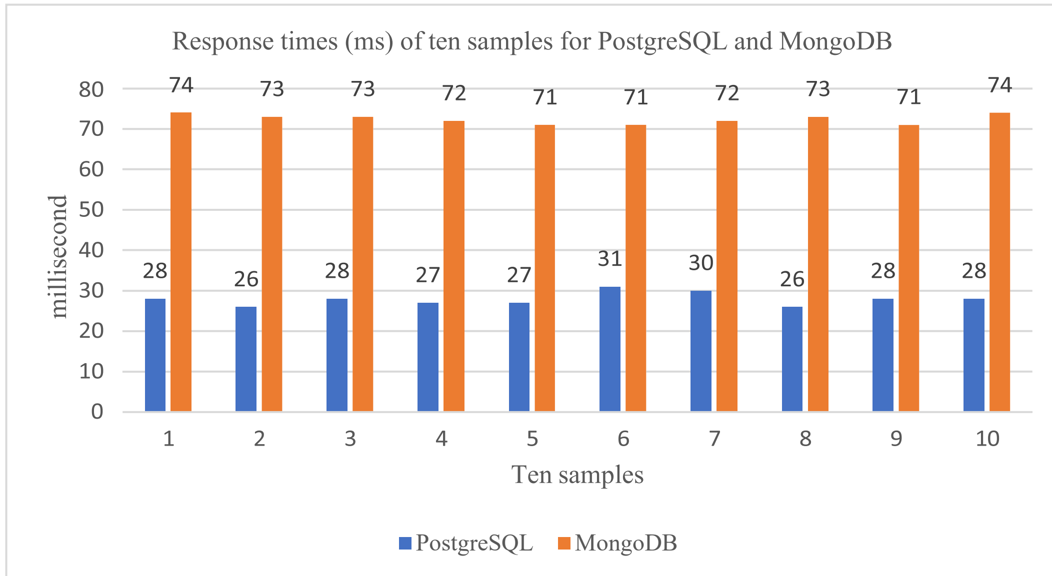

Figure 28: The experiment result of ten samples for updating 1,000 orders in PostgreSQL and MongoDB

Figure 29 presented all response times of ten samples for updating 10,000 records in each database. PostgreSQL had an average response time of 80.4ms across ten samples, while MongoDB had an average response time of 176.6ms, 2.1 times as slow as PostgreSQL.

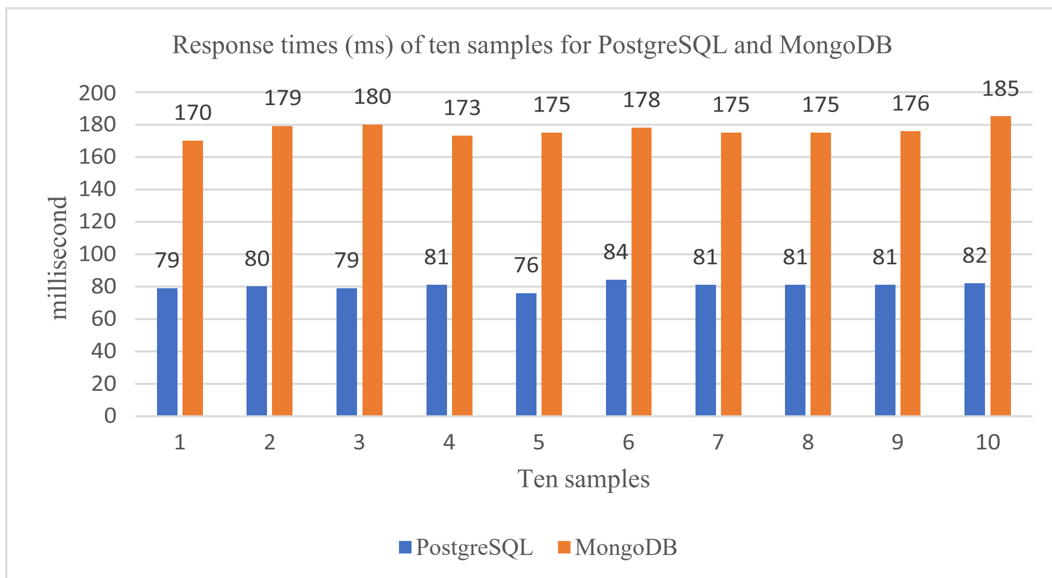

Figure 29: The experiment result of ten samples for updating 10,000 orders in PostgreSQL and MongoDB

Regarding the performance of updating orders, PostgreSQL was faster than MongoDB for updating 100, 1,000 and 10,000 records respectively. The average response time in PostgreSQL was 10ms, while it was 58.9ms in MongoDB when updating 100 records. The average response time in PostgreSQL was 27.9ms, almost 2.6 times as fast as MongoDB when updating 1,000

records. When updating 10,000 records, the average response time in PostgreSQL was 80.4ms, while it reached 176.6ms in MongoDB. This meant that executing update operations in PostgreSQL was more than twice as fast as MongoDB. Generally, the more records the database updated, the longer time the database took. Figure 30 showed that the average response time of each database was getting slower as the number of updating records increased, and it presented the performance comparison for updating 100, 1,000 and 10,000 records between MongoDB and PostgreSQL.

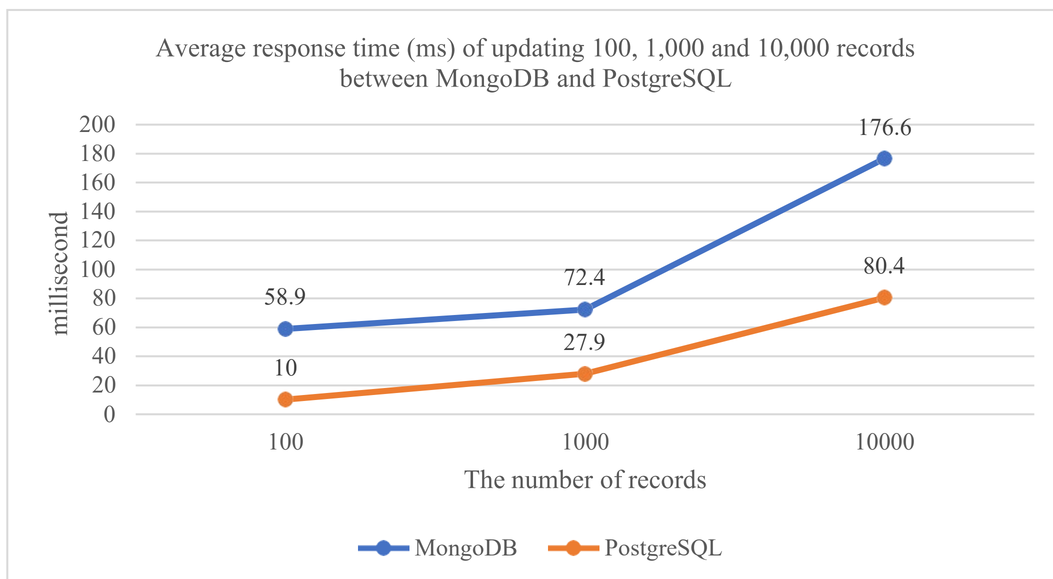

Figure 30: A comparison of average response time for updating different records (100, 1,000 and 10,000) between MongoDB and PostgreSQL

As the number of records increased from 100 to 10,000, the average response time increased by 117.7ms (from 58.9ms to 176.6ms) in MongoDB, while the average response time increased by 70.4ms (from 10ms to 80.4ms) in PostgreSQL. This represented that the increase rate of average response time in MongoDB was bigger than in PostgreSQL for the same scale of growth. As a result, these aspects demonstrated that PostgreSQL performed update operations more efficiently than MongoDB, especially when updating more records.

Scalability Test

This research used the average response time of each user request in a simulated multi-user environment for scalability analysis comparisons between PostgreSQL and MongoDB. There were respectively 60, 120, 240, 480 and 960 concurrent users who updated an order in each database.

Figure 31 illustrated that PostgreSQL got the average response times of 3ms, 3ms, 3ms, 4ms and

4ms, while they were 3ms, 4ms, 4ms, 5ms and 9ms in MongoDB.

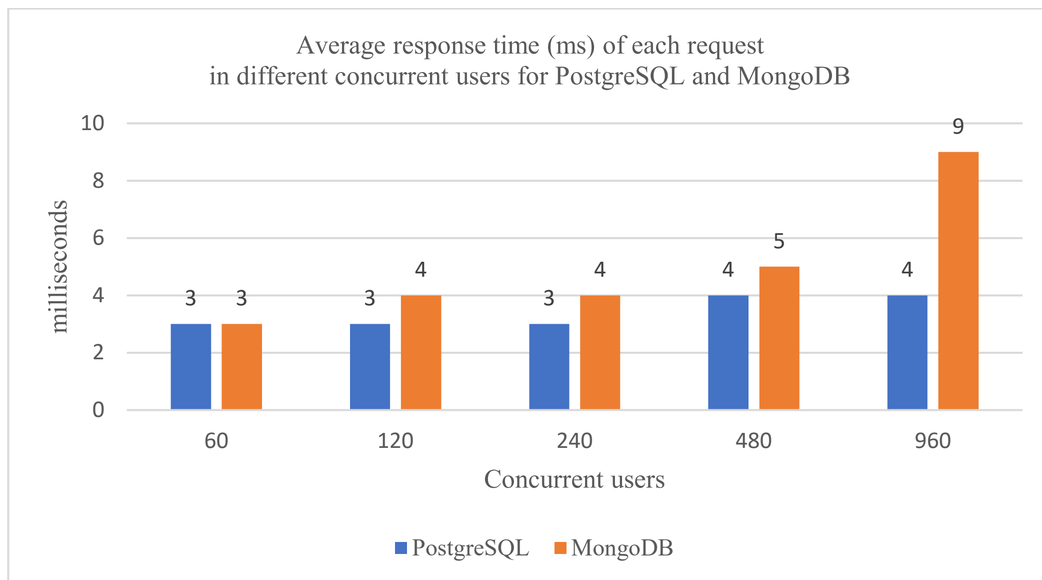

Figure 31: The experiment result of updating an order in a simulated environment with 60, 120, 240, 480 and 960 concurrent users for PostgreSQL and MongoDB

Regarding the scalability performance comparison of updating an order, average response times of the update operation in PostgreSQL were 3ms, 3ms, 3ms, 4ms and 4ms respectively for different numbers of concurrent users (60, 120, 240, 480 and 960), while they were 3ms, 4ms, 4ms, 5ms and 9ms respectively in MongoDB. The result showed that the average response time of each request in PostgreSQL was almost the same as MongoDB for scales of 60, 120, 240 and 480 concurrent users, and both performances were slightly reduced as concurrent users increased. For the performance of the update operation in 960 concurrent requests, the average response time of each request in PostgreSQL was 4ms, twice as fast as MongoDB. However, for MongoDB, the average response time of each request was 9ms with 960 concurrent requests, almost twice slower than the size of 480 concurrent requests. Obviously, as the number of concurrent users was more than 960, MongoDB performance degraded significantly, while PostgreSQL performed stably. The reason was that the throughput of PostgreSQL was becoming much higher than MongoDB when executing 960 concurrent requests. However, there was only a slight difference in throughput between the two databases for sizes of 60, 120, 240, and 480 concurrent requests. Table 13 showed the comparison of throughput for different concurrency levels between

PostgreSQL and MongoDB databases. As a result, for this scalability test, PostgreSQL performed better than MongoDB when user requests were scaled up.

	60	120	240	480	960
PostgreSQL	61.7	128.1	267.9	533.3	1067.9
MongoDB	59.5	125.8	250.5	500.5	991.1

Table 13: A comparison of throughput (requests/sec) for different numbers of concurrent users between PostgreSQL and MongoDB

Order deletion results and findings

Delete Queries

Figure 32 presented all response times of ten samples for deleting 100 records in each database. PostgreSQL had an average response time of 3736.7ms across ten samples, while MongoDB had an average response time of 72.3ms, more than 51 times as fast as PostgreSQL.

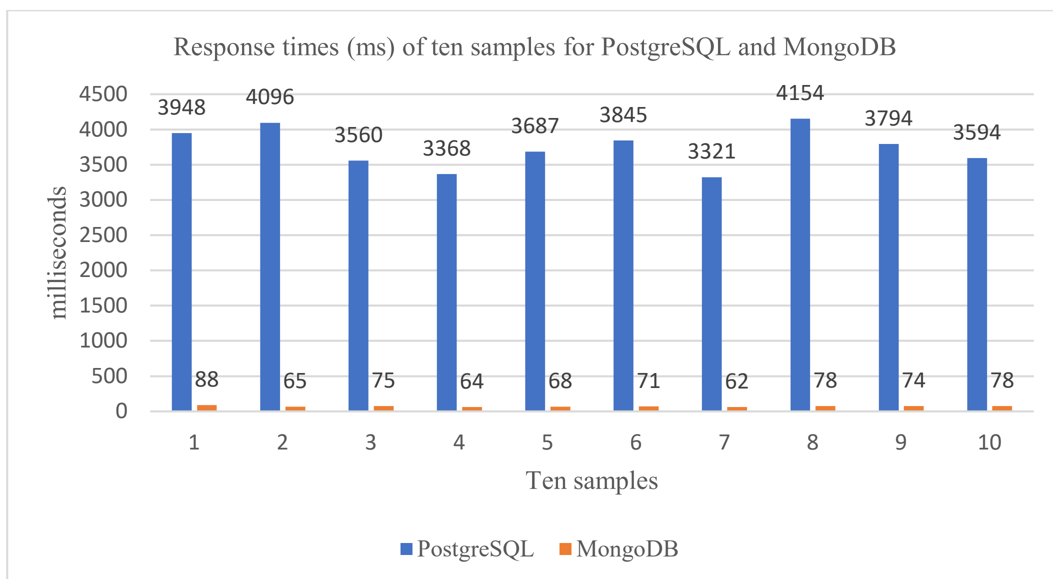

Figure 32: The experiment result of ten samples for deleting 100 historic orders in PostgreSQL and MongoDB

Figure 33 presented all response times of ten samples for deleting 1,000 records in each database. PostgreSQL had an average response time of 32,135.5ms across ten samples, while MongoDB had an average response time of 82.9ms, more than 387 times as fast as PostgreSQL.

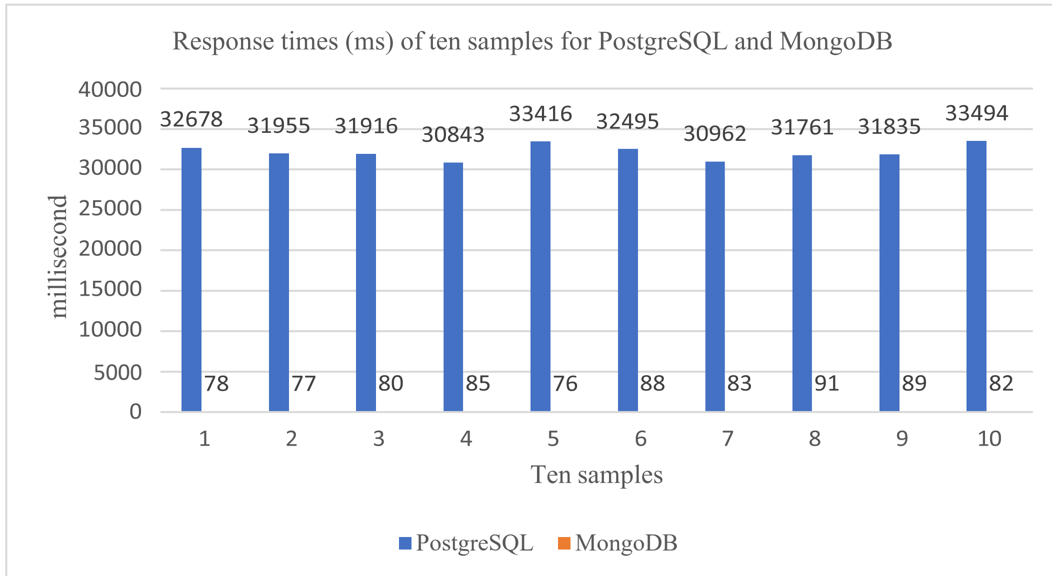

Figure 33: The experiment result of ten samples for deleting 1,000 historic orders in PostgreSQL and MongoDB

Figure 34 presented all response times of ten samples for deleting 10,000 records in each database. PostgreSQL had an average response time of 314,825.5ms across ten samples, while MongoDB had an average response time of 220.7ms, more than 1,426 times as fast as PostgreSQL.

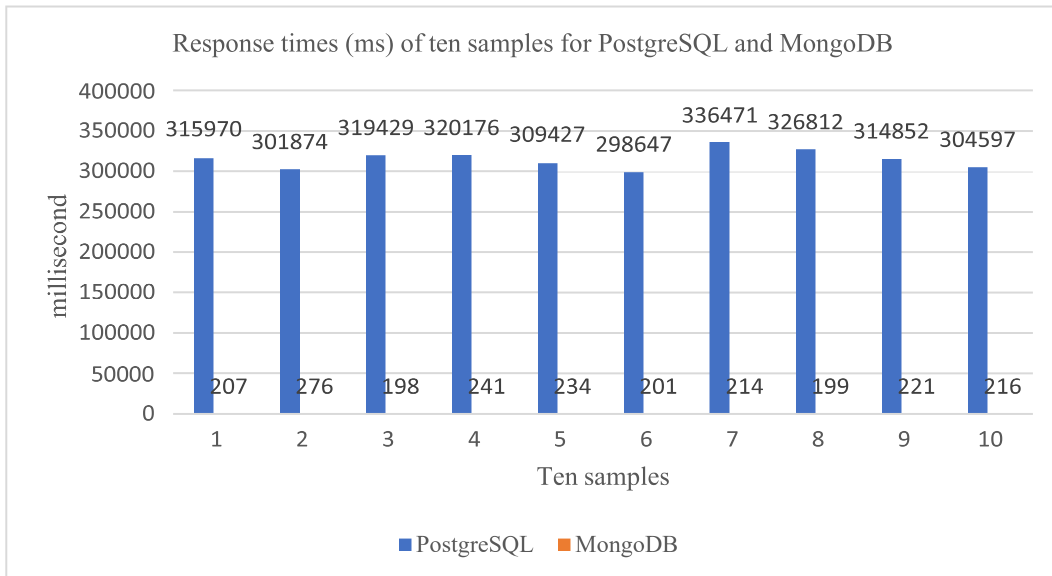

Figure 34: The experiment result of ten samples for deleting 10,000 historic orders in PostgreSQL and MongoDB

Regarding the performance of deleting orders, the average response times in MongoDB was significantly faster than in PostgreSQL for deleting 100, 1,000 and 10,000 records respectively.

The average response time in PostgreSQL was 3,736.7ms, while it was only 72.3ms in MongoDB for deleting 100 records. The average response time in MongoDB was 82.9ms, more than 387 times faster than the 32,135.5ms in PostgreSQL when deleting 1,000 records. When deleting 10,000 records, the average response time in MongoDB was only 220.7ms, while it reached 314,825.5ms in PostgreSQL. This meant that executing delete operations in MongoDB was more than 1,426 times faster than PostgreSQL. In general, the more records the database deleted, the longer time the database took. Figure 35 showed that the average response time of each database was getting longer as the number of records increased, and it presented the performance comparison for deleting 100, 1,000 and 10,000 records between MongoDB and PostgreSQL.

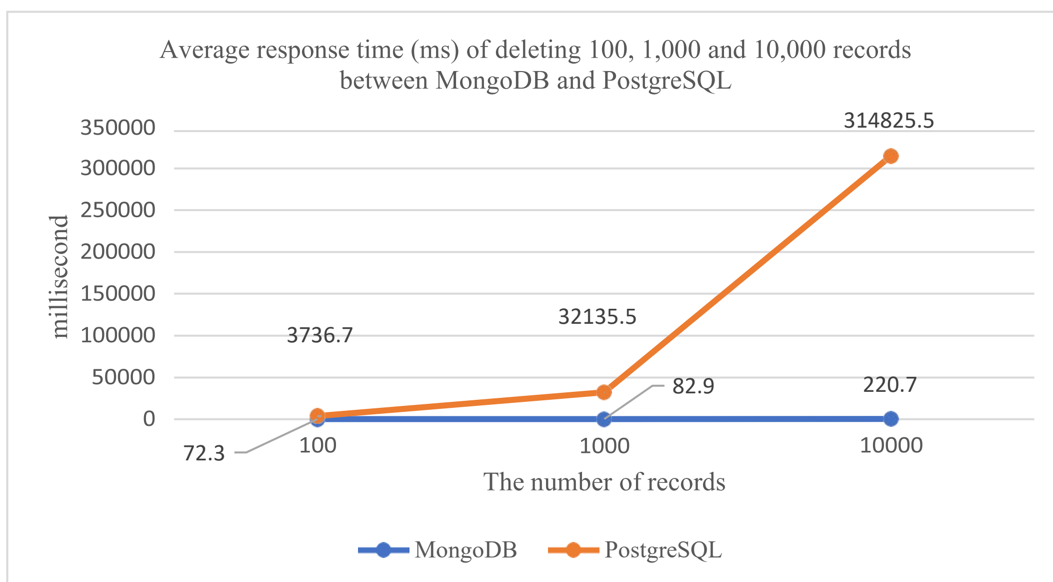

Figure 35: A comparison of average response time
for deleting different records (100, 1,000 and 10,000) between MongoDB and PostgreSQL

In PostgreSQL, the average response time was not only extremely slow at 3,736.7ms at beginning, but as the number of records increased from 100 to 10,000, the average response time increased 84 times and reached 314,825.5ms. In contrast, the average response time only increased by 148.4ms (from 72.3ms to 220.7ms) in MongoDB. This represented that the increase rate of average response time in PostgreSQL significantly greater than in MongoDB for the same scale of growth. As a result, these aspects showed that MongoDB performed delete operations significantly more efficiently than PostgreSQL, especially when deleting a large number of records.

Scalability Test

This research used the average response time of each user's request in a simulated multi-user environment for scalability analysis comparisons between PostgreSQL and MongoDB. There were respectively 60, 120, 240, 480 and 960 concurrent users who deleted an order in each database. Figure 36 illustrated that PostgreSQL got the average response time of 3ms, 4ms, 4ms, 5ms and 5ms, while they were 67ms, 118ms, 701ms, 1,953ms and 4,530ms in MongoDB.

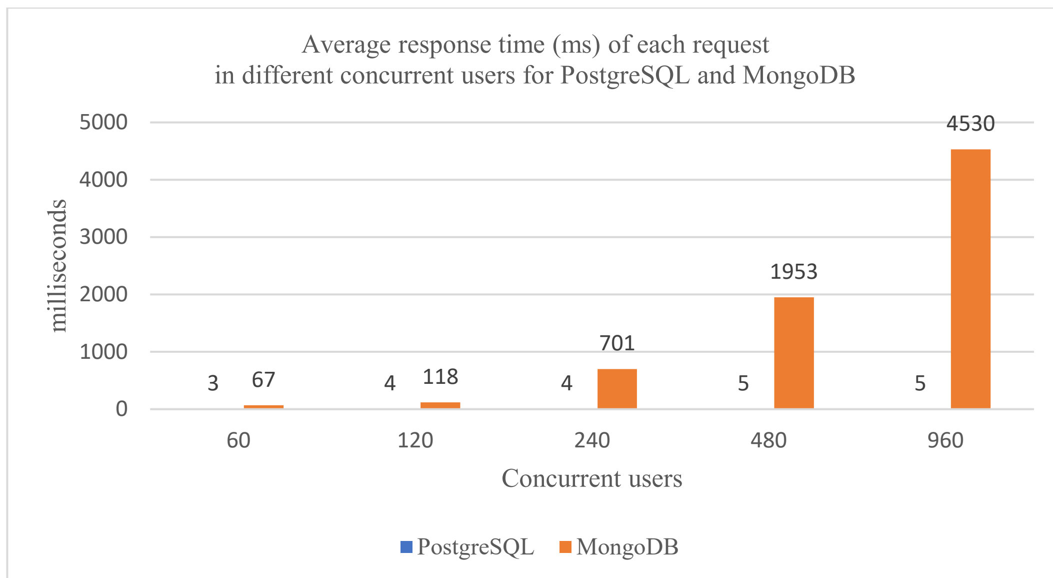

Figure 36: The experiment result of deleting an order in a simulated environment with 60, 120, 240, 480 and 960 concurrent users for PostgreSQL and MongoDB

Regarding the scalability performance comparison of deleting an order, the average response times of the delete operation in PostgreSQL were 3ms, 4ms, 4ms, 5ms and 5ms respectively for different numbers of concurrent users (60, 120, 240, 480 and 960), while they were 67ms, 118ms, 701ms, 1,953ms and 4,530ms respectively in MongoDB. In 60 concurrent users, the average response time in PostgreSQL was almost 22 times faster than MongoDB. For each request in 120 concurrent users, the average response time in PostgreSQL was almost 30 times faster than MongoDB. In 240 concurrent users, the average response time in PostgreSQL was almost 175 times faster than MongoDB. For each request in 480 concurrent users, the average response time in PostgreSQL was almost 390 times faster than MongoDB. This gap increased to 906 times when executing the delete query with 960 concurrent users. Obviously, the performance of each request in MongoDB degraded significantly as the number of concurrent users increased, while the performance in PostgreSQL was not much affected. The reason was that the throughput of

PostgreSQL was doubled as the number of concurrent users increased, while the throughput of MongoDB barely increased when the number of concurrent users exceeded 120. Table 14 showed the comparison of throughput for different concurrency levels between PostgreSQL and MongoDB databases. As a result, PostgreSQL outperformed MongoDB for this scalability testing.

	60	120	240	480	960
PostgreSQL	59.8	126.6	252.1	499.5	1002.1
MongoDB	57.6	115.3	116.3	119.5	122.1

Table 14: A comparison of throughput (requests/sec) for the five different numbers of concurrent users between PostgreSQL and MongoDB

Summary

Regarding the performance results of searching product, both search queries and aggregate queries tests had the same results: MongoDB performed better than PostgreSQL at processing a small number of records (less than or equal to 10,000 records in this experiment), and PostgreSQL was more efficient than MongoDB when processing massive records (100,000 records in this experiment).

In the order placement and the order update result, MongoDB was more efficient than PostgreSQL when executing insert operations, while PostgreSQL performed better when executing update operations. In addition, MongoDB obtained better performance on insert operations than update operations, whereas PostgreSQL performs better on update operations than insert operations. Consequently, MongoDB was more suitable for systems requiring a large amount of insert queries, while PostgreSQL worked better in situations when updating a large number of records is more often the case.

In the order deletion performance test, although MongoDB obtained slower performance on delete operations than update operations, it was still significantly more efficient on deletion than PostgreSQL. Consequently, MongoDB outperformed PostgreSQL on deletion cases.

According to the scalability test results of CRUD operations, PostgreSQL slightly performed better than MongoDB on insert and update operations, while when executing read and delete operations, PostgreSQL performed much faster than MongoDB, especially when the size of concurrent users increased largely. Figure 37 presented the average response times of performing each request under different concurrent requests for PostgreSQL. Obviously, as the number of concurrent requests increased, it had little impacted on the performance of insert, update, and delete operations. However, the performance of read operation degraded by 1 time when the number concurrent request increased from 60 to 960.

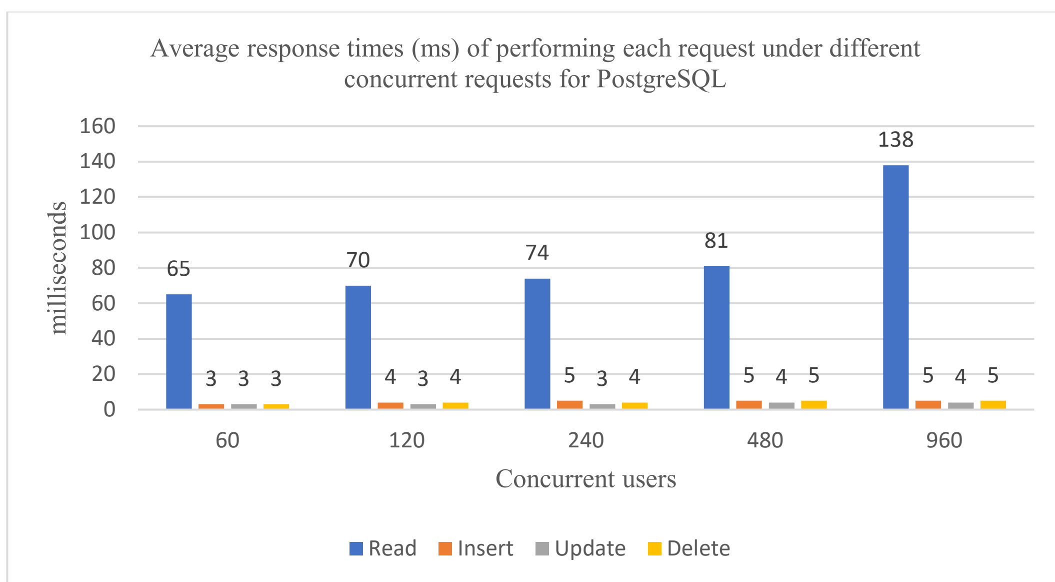

Figure 37: Average response times of performing each request under different concurrent requests for PostgreSQL

Figure 38 showed the average response times of performing each request under different concurrent requests for MongoDB. Apparently, as the number of concurrent requests increased, it had little impacted on the performance in insert and update operations. However, the performance of read and delete operations significantly degraded when the number of concurrent requests increased.

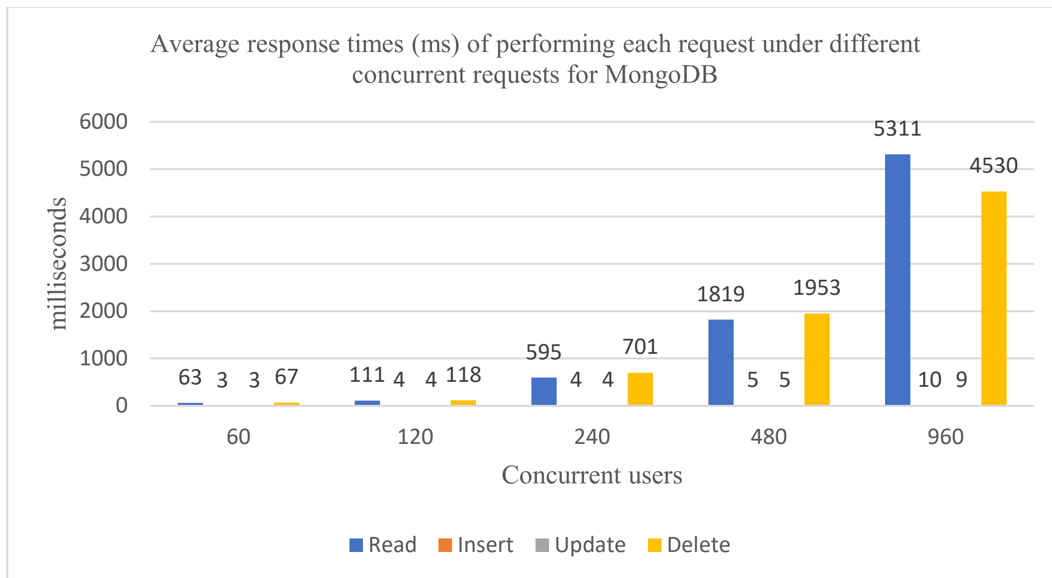

Figure 38: Average response times of performing each request under different concurrent requests for MongoDB

Discussion

According to the obtained performance results reported in the section of experiment results and findings, this section will summarise these results. Obviously, there is no simple way to describe which database performs better, as it really depends on the specific situation and the scale of an application. In this research, performance results were obtained via CRUD operations use cases, and each operation was executed with either single thread or multiple threads. In this section, single thread comparison will be discussed first, followed by multiple threads.

Single-thread

In single thread scenarios, all operations were performed on different numbers of records to verify how the performance was for each database. In this experiment, it was noticed that in single thread scenarios, MongoDB outperformed PostgreSQL when executing both insert and delete operations but lost to the later in update operations.

	Read	Insert	Update	Delete
PostgreSQL	Won @100k	Lost	Won	Lost
MongoDB	Won	Won	Lost	Won

Table 15: Single thread performance comparison between PostgreSQL and MongoDB

Insert and Delete

When inserting 10,000 records into database, MongoDB only took 40.2ms, where PostgreSQL took 1048.3ms, more than 20 times slower than the former. When deleting 10,000 records from database, PostgreSQL performed very badly and took 314825.5ms, while Mongo only used 220.7ms, more than 1430 times faster than the former. This finding echoes with the research result conducted by Jung et al. (2015) who also found that MongoDB performed much better when executing insert and delete operations, especially when the number of records is huge. In their research, they used sample sizes of 30,000, 90,000, 150,000, 210,000 and 300,000, much bigger than this research. Therefore, it is concluded that MongoDB outperformed PostgreSQL in inserting and deleting records in single thread cases.

Search and Aggregate

When executing search queries and aggregate queries, PostgreSQL performed better than MongoDB, even though the latter did perform slightly more efficiently when searching small chunks of products (100, 1,000 and 10,000) from the database. It is also noticed that when the number records increased largely to 100,000, PostgreSQL surpassed MongoDB and performed better. This finding resonates with Jung et al. (2015) again and they noticed that MongoDB performed better when reading 30,000 records but it lost to PostgreSQL when processing a larger number of records like 90,000, 150,000, 210,000 and 300,000. Jung et al. (2015) did not include aggregate queries in their research, while Aboutorabi[a] et al. (2015) did using SQL server and MongoDB. Aboutorabi[a] et al. (2015) also found that SQL performed much better than MongoDB when processing aggregate queries count, sum and avg, but their research did not indicate the volume of data these aggregate queries returned, which could make the result inaccurate as these two databases may perform differently when the returned volume changes.

Update

When updating records into database, this research found PostgreSQL was faster with an average response time of 80.4ms to process 10,000 records, whereas MongoDB used 176.6ms to process the

same number of records, twice slower of the former. Surprisingly, this result was opposed to the results of et al. (2015) who found MongoDB still performed better with update operations with larger number of records till 3 million. As Jung et al. (2015) did not include the details of their update queries, it is difficult to compare the difference between their update queries used with this research. One big difference between these two research studies is the different type of data used, where Jung et al. (2015) used user mileage data with a much simpler data model containing only 3 tables, while this research used ecommerce data with a more complex data model containing 5 tables.

In real ecommerce scenarios like in Amazon and Taobao, end users (product seekers) normally would not update, insert, or delete large numbers of products into/from an order, unless the user is a system admin. Instead, these users would use search and aggregate search (e.g., count search result) functions more, when searching products from the ecommerce system. Therefore, even though MongoDB won three out of the four CRUD operations in single thread scenarios, the author still thinks PostgreSQL would be a better fit for a real ecommerce system.

Multi-threads

Comparing to single thread, a multi-thread test is more important to big ecommerce platforms like Amazon and Taobao, especially during peak seasons like Taobao double 11 and Amazon Prime Day. If the system cannot perform stably and even crashes during peak time, it would cost the platform millions of dollars like the famous Amazon prime day crash in 2018 (Roessel, 2018). In the experiment tests conducted in this research, it was noticed that PostgreSQL outperformed MongoDB in all four CRUD scalability tests, with the former achieved a big win in search and delete operations and performed slightly quicker in insert and update operations. The researcher also studied the throughput of these two databases when executing these multi-thread operations and noticed that PostgreSQL had a much bigger throughput in read and delete operations and a slightly bigger throughput in insert and update operations. It was also found that when the number of concurrent requests increased, the performance and throughput difference became bigger among PostgreSQL and MongoDB.

Flores et al. (2018) performed a similar concurrent test in SQL server and MongoDB with 99 concurrent requests and noticed that the former performed better than the later in Windows environment, so they added a similar test in Linux environment and found MongoDB on Linux performed better than PostgreSQL on windows. It is a pity that they did not include a test with PostgreSQL in Linux, so it would be unfair to conclude that MongoDB performed better than SQL. In contrast, the performance comparison conducted by this research was in the same operation system running in the same set of hardware with the same sample scales of 60, 120, 240, 480 and 960 in four CRUD operations. Therefore, the conclusion that PostgreSQL outperformed MongoDB in multi-thread scenarios is unbiased and fair comparing to Flores et al. (2018)

Summary

This research did not find a specific database performed substantially better than the other in single thread scenarios but did find that SQL represented by PostgreSQL is a better choice for a big ecommerce platform where hundreds of concurrent requests may be coming through at the same time, for example the double 11 festival of Taobao and the prime day of Amazon. On the other hand, it is also a trend that big ecommerce systems started to adopt a combination of NoSQL and SQL as microservice architectures become increasingly popular. In these architectures, platforms contain a collection of services covering various parts of the platform like chat function, product search, customer management, order placement, delivery arrangement and product review, and each service may contain its own application and appropriate database. For the essential parts of the ecommerce system, where data may be more relational rather than standalone document like, it is recommended to use PostgreSQL as it is more efficient to process concurrent requests and relations among multiple tables. For parts where data is more standalone document like, for example product review and chat functions, developers can consider adapting a NoSQL database like MongoDB as it performs better in inserting and updating large amounts of data.

Conclusion

To answer the main research question "Is NoSQL database more efficient than SQL database for different sizes of ecommerce systems?", three sub-questions and hypotheses were raised.

Hypothesis 1: NoSQL will outperform SQL on small ecommerce platforms.

Yes, in 4 of 5 single thread tests (search, aggregate, insert and delete).

No, in 1 of 5 single thread tests (update).

Hypothesis 2: NoSQL will outperform SQL on big ecommerce platforms.

Yes, in 2 of 5 single thread tests (insert and delete).

No, in 3 of 5 single thread tests (search, aggregate and update).

Hypothesis 3: NoSQL will outperform SQL on high concurrency situation.

No, in all five scalability tests (multiple thread tests).

Based on results listed above, three hypotheses failed, which leads to the answer to the main research question "Is NoSQL database more efficient than SQL database for different sizes of ecommerce systems?": NO.

Therefore, this research found (in the single node test environment) PostgreSQL is a better fit for ecommerce systems, maybe because the data used for this research is more "relation" based, rather than stand-alone documents.

NoSQL databases like MongoDB may be not suitable to be used in the essential sell-and-buy activities of an ecommerce system, as these activities are more relational. But ecommerce systems that encounter scalability issues should consider using NoSQL for its non-relational parts of the system, e.g., chat and review functions.

Limitations

Because this research did not seek research funds, the author can only use open-source databases PostgreSQL and MongoDB and home-grade hardware environment for performance tests. As a result, the test results cannot be accurately reflected on similar projects in an enterprise-grade. To thoroughly compare the performance of a SQL and a NoSQL database, these important NoSQL features should have been included: the ease of scaling horizontally (e.g., data can be replicated and portioned across multiple servers), auto sharding, maximizing disk space and dynamically loading balance queries, because these features would have improved NoSQL performance significantly. Unfortunately, as the researcher could not access a better testing hardware or environment, the above-mentioned features were not included in this performance test. Additionally, this research project is the first research that the author has ever conducted, so there is a limit in professional experience throughout this research, such as designing experiments and scaling of sizes.

Future research

Due to limitations of research budget and research support the author could gain access to, this research was performed on a single node home-grade hardware environment and only used batches of 100, 1,000 and 10,000 for the performance test. Future research that can access a better budget and support should consider conducting this type of performance test with an enterprise-grade hardware setup, multiple nodes and much larger batches. In addition, as mentioned above, this performance comparison research did not include important NoSQL features, due to some objective factors. Future research is suggested to include testing sharding and partitioning features in distributed servers when comparing the performance of SQL and NoSQL databases. On the other hand, other relational databases like SQL Server, Oracle and MySQL can be used to represent SQL, while other types of NoSQL databases like key-value store, column store and graph store can be involved. In addition, an ecommerce system usually consists of different modules, so future research can study which database is a better fit for which ecommerce module.

References

Aatish, C. (2017, 19-21 July 2017). *Automating the process from E-Commerce to M-Commerce.* Paper presented at the 2017 1st International Conference on Next Generation Computing Applications (NextComp).

Aboutorabi[a], S. H., Rezapour, M., Moradi, M., & Ghadiri, N. (2015, 18-19 Aug. 2015). *Performance evaluation of SQL and MongoDB databases for big e-commerce data.* Paper presented at the 2015 International Symposium on Computer Science and Software Engineering (CSSE).

Abramova, V., & Bernardino, J. (2013). *NoSQL databases: MongoDB vs cassandra.* Paper presented at the Proceedings of the international C* conference on computer science and software engineering.

ACM. (2014). For fundamental contributions to the concepts and practices underlying modern database systems. Retrieved from https://amturing.acm.org/award_winners/stonebraker_1172121.cfm

Al-Tit, A. A. (2020). E-commerce drivers and barriers and their impact on e-customer loyalty in small and medium-sized enterprises (SMES). *Business: Theory & Practice, 21*(1), 146-157. doi:10.3846/btp.2020.11612

Amazon Web Services. (n.d.). What is a relational database? Retrieved from https://aws.amazon.com/relational-database/

Andlinger, P. (2013). RDBMS dominate the database market, but NoSQL systems are catching up. Retrieved from https://db-engines.com/en/blog_post/23

AWS. (n.d.-a). What is a document database? Retrieved from https://aws.amazon.com/nosql/document/

AWS. (n.d.-b). What is a key-value database? Retrieved from https://aws.amazon.com/nosql/key-value/

Banerjee, S., Goto, T., Debnath, N. C., & Sarkar, A. (2017, 24-26 July 2017). *Ontology driven query language for NoSQL databases.* Paper presented at the 2017 IEEE 15th International Conference on Industrial Informatics (INDIN).

Barkho, G. (2021). How Facebook is doubling down on Marketplace. Retrieved from https://www.modernretail.co/platforms/facebook-marketplace-pandemic-growth/

Batra, R. (2018). A History of SQL and Relational Databases. In *SQL Primer: An Accelerated Introduction to SQL Basics* (pp. 183-187). Berkeley, CA: Apress.

Berners-Lee, T., Dimitroyannis, D., Mallinckrodt, A. J., & McKay, S. (1994). World Wide Web. *Computers in Physics, 8*(3), 298-299. doi:10.1063/1.4823300

Bhogal, J., & Choksi, I. (2015). *Handling big data using NoSQL.* Paper presented at the 2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops.

Chand, M. (2019). What Is A Network Database. Retrieved from https://www.c-sharpcorner.com/article/what-is-a-network-database/

Chandra, D. G. (2015). BASE analysis of NoSQL database. *Future Generation Computer Systems, 52*, 13-21.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., . . . Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS), 26*(2), 1-26.

Clement, J. (2020). Total and e-commerce U.S. retail trade sales 2000-2018. Retrieved from https://www.statista.com/statistics/185283/total-and-e-commerce-us-retail-trade-sales-since-2000/

Clement, J. (2021). *Most popular online retail websites worldwide in 2020, by average monthly traffic*. Retrieved from https://www.statista.com/statistics/274708/online-retail-and-auction-ranked-by-worldwide-audiences/

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM, 13*(6), 377-387. Retrieved from https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

Codd, E. F. (1985). Is your DBMS really relational. *Computerworld, 14*(10).

Codd, E. F. (1989). Relational database: A practical foundation for productivity. In *Readings in Artificial Intelligence and Databases* (pp. 60-68): Elsevier.

Computer World. (2017). Ransomware groups have deleted over 10,000 MongoDB databases. Retrieved from https://www.computerworld.com/article/3155260/ransomware-groups-have-deleted-over-10000-mongodb-databases.html

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., . . . Hochschild, P. (2013). Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS), 31*(3), 1-22.

Costello, K. (2019). Gartner Says the Future of the Database Market Is the Cloud. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2019-07-01-gartner-says-the-future-of-the-database-market-is-the

Darwen, H. (1996). Business System 12 (BS12). Retrieved from https://www.mcjones.org/System_R/bs12.html

Data Commons. (2019). United States of America population. Retrieved from https://datacommons.org/place/country/USA?utm_medium=explore&mprop=count&popt=Person&hl=en

Date, C. J. (2005). *Database in depth: relational theory for practitioners*: " O'Reilly Media, Inc.".

Davis, E. (2016). Understanding 4 Database Types, for Ecommerce. Retrieved from https://www.practicalecommerce.com/Understanding-4-Database-Types-for-Ecommerce

DB-Engines. (2021a). DB-Engines Ranking. Retrieved from https://db-engines.com/en/ranking

DB-Engines. (2021b). DB-Engines Ranking - Trend Popularity. Retrieved from https://db-engines.com/en/ranking_trend

DB-Engines. (2021c). DBMS popularity broken down by database model. Retrieved from https://db-engines.com/en/ranking_categories

De Angelis, S., Aniello, L., Baldoni, R., Lombardi, F., Margheri, A., & Sassone, V. (2018). PBFT vs proof-of-authority: Applying the CAP theorem to permissioned blockchain. Retrived from https://eprints.soton.ac.uk/415083/

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., . . . Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review, 41*(6), 205-220.

DeWitt, D. J. (1993). The Wisconsin Benchmark: Past, Present, and Future. In.

ecommerceDB.com. (2019). *In-depth: B2B e-Commerce 2019*. Retrieved from https://www.statista.com/study/44442/statista-report-b2b-e-commerce/

EDUCBA. (n.d.). Introduction to Hierarchical Database Model. Retrieved from https://www.educba.com/hierarchical-database-model/

Everts, T. (2016). Mobile Load Time and User Abandonment. Retrieved from https://developer.akamai.com/blog/2016/09/14/mobile-load-time-user-abandonment

Facebook Engineering. (2008). Cassandra – A structured storage system on a P2P Network. Retrieved from https://www.facebook.com/notes/facebook-engineering/cassandra-a-structured-storage-system-on-a-p2p-network/24413138919/

FairCom. (2018). Database Talk: What is ACID compliance? Retrieved from https://www.faircom.com/insights/database-talk-acid-compliance

Fandom. (n.d). Network database model. Retrieved from https://databasemanagement.fandom.com/wiki/Network_Database_Model

Flores, A., Ramírez, S., Toasa, R., Vargas, J., Urvina-Barrionuevo, R., & Lavin, J. M. (2018). *Performance Evaluation of NoSQL and SQL Queries in Response Time for the E-government.* Paper presented at the 2018 International Conference on eDemocracy & eGovernment (ICEDEG).

Garcia, D. F., & Garcia, J. (2003). TPC-W e-commerce benchmark evaluation. *Computer, 36*(2), 42-48. doi:10.1109/MC.2003.1178045

GeeksforGeeks. (2021). Advantages and Disadvantages of SQL. Retrieved from https://www.geeksforgeeks.org/advantages-and-disadvantages-of-sql/

Gefen, D., & Straub, D. (2003). Managing User Trust in B2C e-Services. *e-Service Journal, 2*(2), 7-24. doi:10.2979/esj.2003.2.2.7

German, D. M. (2006). *A study of the contributors of PostgreSQL.* Paper presented at the Proceedings of the 2006 international workshop on Mining software repositories.

Ghazal, A., Ivanov, T., Kostamaa, P., Crolotte, A., Voong, R., Al-Kateb, M., . . . Zicari, R. V. (2017, 19-22 April 2017). *BigBench V2: The New and Improved BigBench.* Paper presented at the 2017 IEEE 33rd International Conference on Data Engineering (ICDE).

Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., & Jacobsen, H.-A. (2013). *Bigbench: Towards an industry standard benchmark for big data analytics.* Paper presented at the Proceedings of the 2013 ACM SIGMOD international conference on Management of data.

González-Aparicio, M. T., Younas, M., Tuya, J., & Casado, R. (2016). *A new model for testing CRUD operations in a NoSQL database.* Paper presented at the 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA).

Gray, J. (1993). Database and Transaction Processing Performance Handbook. Retrieved from http://jimgray.azurewebsites.net/benchmarkhandbook/chapter1.pdf

Grolinger, K., Higashino, W. A., Tiwari, A., & Capretz, M. A. (2013). Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: advances, systems and applications, 2*(1), 22.

Győrödi, C., Győrödi, R., Pecherle, G., & Olah, A. (2015). *A comparative study: MongoDB vs. MySQL.* Paper presented at the 2015 13th International Conference on Engineering of Modern Electric Systems (EMES).

Habib, O. (2015). A Newbie Guide to Databases. Retrieved from https://www.appdynamics.com/blog/engineering/a-newbie-guide-to-databases/

Harizopoulos, S., Abadi, D., & Boncz, P. (2009). Column-Oriented Database Systems. Retrieved from http://www.cs.umd.edu/~abadi/talks/Column_Store_Tutorial_VLDB09.pdf

Hecht, R., & Jablonski, S. (2011). *NoSQL evaluation: A use case oriented survey.* Paper presented at the 2011 International Conference on Cloud and Service Computing.

Himango, J. (2017). The Biggest Challenges of Moving to NoSQL. Retrieved from https://dzone.com/articles/the-biggest-challenges-of-moving-to-nosql

Homan, J. V., & Kovacs, P. J. (2009). A comparison of the relational database model and the associative database model. *Issues in Information Systems, 10*(1), 208-213.

IBM Knowlwdge Center. (2020). ACID properties of transactions. Retrieved from https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.4.0/product-overview/acid.html

Ihde, N., Marten, P., Eleliemy, A., Poerwawinata, G., Silva, P., Tolovski, I., . . . Rabl, T. (2021). A Survey of Big Data, High Performance Computing, and Machine Learning Benchmarks.

Internet World Stats. (2021). World internet users and 2021 population stats. Retrieved from https://www.internetworldstats.com/stats.htm

Jiang, S. (2020). Type of E-Commerce. In ecommerce.asp (Ed.).

JMeter, A. (n.d.). Overview. Retrieved from https://jmeter.apache.org/index.html

Jung, M., Youn, S., Bae, J., & Choi, Y. (2015, 25-28 Nov. 2015). *A Study on Data Input and Output Performance Comparison of MongoDB and PostgreSQL in the Big Data Environment.* Paper presented at the 2015 8th International Conference on Database Theory and Application (DTA).

Jutla, D., Bodorik, P., & Wang, Y. (1999). Developing internet e-commerce benchmarks. *Information Systems, 24*(6), 475-493.

Kaggle. (n.d.-a). Brazilian E-Commerce Public Dataset by Olist. Retrieved from https://www.kaggle.com/olistbr/brazilian-ecommerce

Kaggle. (n.d.-b). Terms. Retrieved from https://www.kaggle.com/terms

Kenton, W. (2021). Business-to-consumer (B2C). Retrieved from https://www.investopedia.com/terms/b/btoc.asp

Keshavarz, S. (2021). Analyzing Performance Differences Between MySQL and MongoDB.

Kim, C., & Shim, K. (2015). Supporting set-valued joins in NoSQL using MapReduce. *Information Systems, 49*, 52-64. doi:https://doi.org/10.1016/j.is.2014.11.005

Klostermeyer, P. (2021). *Performance Benchmarking of NewSQL Databases with Yahoo Cloud Serving Benchmark.* Paper presented at the Proceedings of the Future Technologies Conference (FTC) 2020, Volume 2.

Krebson Security. (2017). Extortionists Wipe Thousands of Databases, Victims Who Pay Up Get Stiffed. Retrieved from https://krebsonsecurity.com/2017/01/extortionists-wipe-thousands-of-databases-victims-who-pay-up-get-stiffed/

Kroenke, D. (1977). *Database Processing, Fundamentals, Modeling, Applications*. Chicago: Science Research Associates.

Kunda, D., & Phiri, H. (2017). A comparative study of nosql and relational database. *Zambia ICT Journal, 1*(1), 1-4.

Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review, 44*(2), 35-40.

Li, Y., & Manoharan, S. (2013). *A performance comparison of SQL and NoSQL databases.* Paper presented at the 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM).

Liu, X. (2015). *An Analysis of Relational Database and NoSQL Database on an Ecommerce Platform.* Trinity College,

Ma, Y. (2020a). Number of online shoppers in China 2009-2020. Retrieved from https://www.statista.com/statistics/277391/number-of-online-buyers-in-china/

Ma, Y. (2020b). Online shopping market gross merchandise volume in China 2013-2022. Retrieved from https://www.statista.com/statistics/278555/china-online-shopping-gross-merchandise-volume/

Makris, A., Tserpes, K., Spiliopoulos, G., & Anagnostopoulos, D. (2019). *Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data.* Paper presented at the EDBT/ICDT Workshops.

Maria DB. (n.d.). Understanding the Network Database Model. Retrieved from https://mariadb.com/kb/en/understanding-the-network-database-model/

Martins, P., Tomé, P., Wanzeller, C., Sá, F., & Abbasi, M. (2021). Comparing Oracle and PostgreSQL, Performance and Optimization. In *World Conference on Information Systems and Technologies* (pp. 481-490). Springer, Cham.

Mathew, A. B., & Kumar, S. M. (2015). *Analysis of data management and query handling in social networks using NoSQL databases.* Paper presented at the 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI).

Menascé, D. A. (2002). TPC-W: A benchmark for e-commerce. *IEEE Internet Computing, 6*(3), 83-87.

Meyer, S. (2020). Understanding the COVID-19 effect on online shopping behavior. Retrieved from https://www.bigcommerce.com/blog/covid-19-ecommerce/#changes-in-revenue-across-ecommerce

Meysman, A. (2016). NoSQL Database Types. Retrieved from https://dzone.com/articles/nosql-database-types-1#:~:text=There%20are%20four%20big%20NoSQL,model%20database%2C%20combining%20NoSQL%20types.

Mohammed, J. (2015a). Is Postgres NoSQL better than MongoDB? Retrieved from https://www.aptuz.com/blog/is-postgres-nosql-database-better-than-mongodb/

Mohammed, J. (2015b). Is Postgres NoSQL better than MongoDB? Retrieved from https://www.aptuz.com/blog/is-postgres-nosql-database-better-than-mongodb/

Momjian, B. (2021). Postgresql performance tuning. *Linux Journal*, 88, 3-9.

MongoDB Documentation. (NA-a). Introduction to MongoDB. Retrieved from https://docs.mongodb.com/manual/introduction/

MongoDB Documentation. (NA-b). Model one-to-many relationships with embedded documents. Retrieved from https://docs.mongodb.com/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/

Moniruzzaman, A., & Hossain, S. A. (2013). Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*.

Mullane, J. V., Peters, M. H., & Bullington, K. E. (2001). Entrepreneurial firms as suppliers in business-to-business e-commerce. *Management Decision*.

Nakamura, M., Tabaru, T., Ujibashi, Y., Hashida, T., Kawaba, M., & Harada, L. (2015, 20-22 May 2015). *Extending postgreSQL to handle OLXP workloads.* Paper presented at the Fifth International Conference on the Innovative Computing Technology (INTECH 2015).

Nambiar, R., & Poess, M. (2013). Keeping the TPC relevant! *Proc. VLDB Endow., 6*(11), 1186–1187. doi:10.14778/2536222.2536252

Nayak, A., Poriya, A., & Poojary, D. (2013). Type of NOSQL databases and its comparison with relational databases. *International Journal of Applied Information Systems, 5*(4), 16-19.

Neo4J. (n.d.-a). Why the most important part of Facebook Graph Search is "Graph". Retrieved from https://neo4j.com/news/facebook-graphsearch/

Neo4J. (n.d.-b). Map functions. Retrieved from https://neo4j.com/labs/apoc/4.1/data-structures/map-functions/

Nevedrov, D. (2006). Using JMeter to Performance Test Web Services. *Published on dev2dev*, 1-11.

Ohyver, M., Moniaga, J. V., Sungkawa, I., Subagyo, B. E., & Chandra, I. A. (2019). The Comparison Firebase Realtime Database and MySQL Database Performance using Wilcoxon Signed-Rank Test. *Procedia Computer Science, 157*, 396-405. doi:https://doi.org/10.1016/j.procs.2019.08.231

OmniSci. (n.d.). Hierarchical Database. Retrieved from https://www.omnisci.com/technical-glossary/hierarchical-database

Oracle. (2007). 1970s Defying Conventional Wisdom. *Profit Magazine, 12*(2), 28. Retrieved from https://www.oracle.com/us/corporate/profit/p27anniv-timeline-151918.pdf

Padhy, R. P., Patra, M. R., & Satapathy, S. C. (2011). RDBMS to NoSQL: reviewing some next-generation non-relational database's. *International Journal of Advanced Engineering Science and Technologies, 11*(1), 15-30.

Pokorny, J. (2013). NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems*.

Postgres Help. (2019). History of PostgreSQL database. Retrieved from https://postgreshelp.com/history-of-postgresql-database/

RAIMA. (n.d.). Network Database Model Vs. Relational Model. Retrieved from https://raima.com/network-model-vs-relational-model/

Ramakrsihnan, R., Donjerkovic, D., Ranganathan, A., Beyer, K. S., & Krishnaprasad, M. (1998). *Srql: Sorted relational query language.* Paper presented at the Proceedings. Tenth International Conference on Scientific and Statistical Database Management (Cat. No. 98TB100243).

Rautmare, S., & Bhalerao, D. (2016). *MySQL and NoSQL database comparison for IoT application.* Paper presented at the 2016 IEEE International Conference on Advances in Computer Applications (ICACA).

Reynolds, J. (2000). eCommerce: a critical review. *International Journal of Retail & Distribution Management, 28*(10), 417-444. doi:10.1108/09590550010349253

Roessel, L. (2018). Amazon's prime day crash: A lesson in scalability. Retrieved from https://www.sana-commerce.com/blog/amazons-prime-day-crash-lesson-scalability/

SAP Help Portal. (n.d). Multiversion Concurrency Control (MVCC) Issues. Retrieved from https://help.sap.com/viewer/bed8c14f9f024763b0777aa72b5436f6/1.0.12/en-US/94fc07fbeff1474aa878737f2c9921d3.html-

Schmid, S., Galicz, E., & Reinhardt, W. (2015, 19-21 May 2015). *WMS performance of selected SQL and NoSQL databases.* Paper presented at the International Conference on Military Technologies (ICMT) 2015.

Sebora, T. C., Lee, S. M., & Sukasame, N. (2009). Critical success factors for e-commerce entrepreneurship: an empirical study of Thailand. *Small Business Economics, 32*(3), 303-316.

Shopify. (n.d.). About us: Shopify powers over 1,700,000 businesses worldwide. Retrieved from https://www.shopify.co.nz/about

Similarweb. (n.d.-a). Alibaba.com May 2021 overview. Retrieved from https://www.similarweb.com/website/alibaba.com/

Similarweb. (n.d.-b). Tradme.co.nz May 2021 overview. Retrieved from https://www.similarweb.com/website/trademe.co.nz/

Similarweb. (n.d.-c). Amazon.com May 2021 overview. Retrieved from https://www.similarweb.com/website/amazon.com/

Simsek, G. (2019). What is new about NewSQL? Retrieved from https://softwareengineeringdaily.com/2019/02/24/what-is-new-about-newsql/

Singh, S. (2018). Understanding the CAP theorem. Retrieved from https://www.linkedin.com/pulse/understanding-cap-theorem-sanjeev-singh

Song, I.-Y., & Whang, K.-Y. (2000). Database design for real-world e-commerce systems. *IEEE Data Eng. Bull., 23*(1), 23-28.

Stack Overflow. (n.d). Where does mongodb stand in the CAP theorem? Retrieved from https://stackoverflow.com/questions/11292215/where-does-mongodb-stand-in-the-cap-theorem

Statista. (2021). *ECommerce report 2021*. Retrieved from https://www.statista.com/study/42335/ecommerce-report/

Stonebraker, M., & Rowe, L. A. (1986). The design of Postgres. *ACM Sigmod Record, 15*(2), 340-355.

Sumathi, S., & Esakkirajan, S. (2007). Relational Model. In *Fundamentals of Relational Database Management Systems* (pp. 65-102). Berlin, Heidelberg: Springer Berlin Heidelberg.

Takanobu, R., Zhuang, T., Huang, M., Feng, J., Tang, H., & Zheng, B. (2019, May). Aggregating e-commerce search results from heterogeneous sources via hierarchical reinforcement learning. In The World Wide Web Conference (pp. 1771-1781).

TechOpedia. (n.d.). Network Database. Retrieved from https://www.techopedia.com/definition/20971/network-database

Trade on Taobao (n.d.). What is Taobao. Retrieved from http://tradeontaobao.com/what-is-taobao/

Treviño Villalobos, M., Víquez Acuña, L., & Quirós Oviedo, R. (2020). Comparison of the Response Times of MongoDB and PostgreSQL According to Type of Query in Geographical Databases. *Computación y Sistemas, 24*(4).

Truica, C.-O., Boicea, A., & Trifan, I. (2013). *CRUD operations in MongoDB.* Paper presented at the 2013 International Conference on Advanced Computer Science and Electronics Information (ICACSEI 2013).

Ubuntu. (n.d.). Package Management. Retrieved from https://ubuntu.com/server/docs/package-management

Vogels, W. (2009). Eventually consistent. *Communications of the ACM, 52*(1), 40-44.

VMware. (n.d). Defining Database Performance. Retrieved from https://gpdb.docs.pivotal.io/43330/admin_guide/perf_intro.html

Wade, B. W., & Chamberlin, D. D. (2012). IBM relational database systems: The early years. *IEEE Annals of the History of Computing, 34*(4), 38-48.

Weeldreyer, J. A., & Friesen, O. (2001). Multics Relational Data Store (MRDS). *ACM SIGMOD Anthology, 5*.

Wikipedia. (n.d.-a). Network model. Retrieved from https://en.wikipedia.org/wiki/Network_model

Wikipedia. (n.d.-b). Object database. Retrieved from https://en.wikipedia.org/wiki/Object_database#cite_note-USDT01-1

Wikipedia. (n.d.-c). Relational Database. Retrieved from https://en.wikipedia.org/wiki/Relational_database#cite_note-7

Wikipedia. (n.d.-d). MongoDB. Retrieved from https://en.wikipedia.org/wiki/MongoDB#Technical_criticisms

Woodford, C. (2005). *The Internet: a historical encyclopedia* (Vol. 2): ABC-CLIO.

Yan, C., Cheung, A., Yang, J., & Lu, S. (2017). *Understanding database performance inefficiencies in real-world web applications.* Paper presented at the Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.

Yoon, B.-H., Kim, S.-K., & Kim, S.-Y. (2017). Use of Graph Database for the Integration of Heterogeneous Biological Data. *Genomics & informatics, 15*(1), 19-27. doi:10.5808/GI.2017.15.1.19

Zaied, A. N. H. (2012). Barriers to e-commerce adoption in Egyptian SMEs. *International Journal of Information Engineering and Electronic Business, 4*(3), 9.

Zaman, F. U., Khuhro, M. A., Kumar, K., Mirbahar, N., Khan, M. Z., & Kalhoro, A. (2021). Comparative Case Study Difference Between Azure Cloud SQL and Atlas MongoDB NoSQL Database. *International Journal, 9*(7).

Zhang, C., Lu, J., Xu, P., & Chen, Y. (2019, 2019//). *UniBench: A Benchmark for Multi-model Database Management Systems.* Paper presented at the Performance Evaluation and Benchmarking for the Era of Artificial Intelligence, Cham.

Zutshi, R. (1999). Measuring relational database server transaction speeds with the AS3AP benchmark. [Master's thesis, University of North Carolina]. Carolina Digital Repository https://doi.org/10.17615/wv15-v797

Appendix

1 MongoDB connection script

```
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoClient;
import com.mongodb.MongoClientSettings;
import com.mongodb.ServerAddress;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import java.util.Arrays;

try {
  MongoClientSettings settings = MongoClientSettings.builder()
    .applyToClusterSettings {builder ->
      builder.hosts(Arrays.asList(new ServerAddress("localhost",
    27017)))}
    .build();

  MongoClient mongoClient = MongoClients.create(settings);
  MongoDatabase mongoDB = mongoClient.getDatabase("
    brazilian_ecommerce");
  MongoCollection<Document> olist_products_dataset = mongoDB.
    getCollection("olist_products_dataset");
  MongoCollection<Document> olist_orders_dataset = mongoDB.
    getCollection("olist_orders_dataset");
  MongoCollection<Document> olist_order_items_dataset = mongoDB.
    getCollection("olist_order_items_dataset");
  vars.putObject("olist_products_dataset", olist_products_dataset);
  vars.putObject("olist_orders_dataset", olist_orders_dataset);
  vars.putObject("olist_order_items_dataset",
    olist_order_items_dataset);
  return "Connected to MongoDB";
}
catch (Exception e) {
  SampleResult.setSuccessful(false);
  SampleResult.setResponseCode("500");
  SampleResult.setResponseMessage("Exception: " + e);
}
```

2 MongoDB - importing order items in C-Sharp

```
        static void Main(string[] args)
        {
```

```
3           MongoClient mongoClient = new MongoClient("mongodb://localhost:27017");
4           var mongoDatabase = mongoClient.GetDatabase("brazilian_ecommerce");
5           var ordersCollection = mongoDatabase.GetCollection<BsonDocument>("olist_orders_dataset");
6           string postgresConnString = "Server=localhost;Username=postgres;Database=brazilian_ecommerce;Port=5432;Password=Ls0706;SSLMode=Prefer";
7
8           List<OrderItem> orderItems = new List<OrderItem>();
9           using (var conn = new NpgsqlConnection(postgresConnString))
10          {
11              conn.Open();
12              using (var command = new NpgsqlCommand("select o.order_id as id,oi.* from olist_orders_dataset o left join olist_order_items_dataset oi on oi.order_id=o.order_id", conn))
13              {
14                  var reader = command.ExecuteReader();
15                  while (reader.Read())
16                  {
17                      orderItems.Add(new OrderItem()
18                      {
19                          id = reader.GetGuid(0),
20                          product_id = reader.IsDBNull(1) ? null : reader.GetGuid(1),
21                          seller_id = reader.IsDBNull(2) ? null : reader.GetGuid(2),
22                          shipping_limit_date = reader.IsDBNull(3) ? null : reader.GetDateTime(3),
23                          price = reader.IsDBNull(4) ? null : reader.GetDecimal(4),
24                          freight_value = reader.IsDBNull(5) ? null : reader.GetDecimal(5),
25                          order_id = reader.IsDBNull(6) ? null : reader.GetGuid(6),
26                          order_item_id = reader.IsDBNull(7) ? null : reader.GetGuid(7)
27                      });
28                  }
29                  reader.Close();
30              }
31          }
32
33          var groupList = orderItems.GroupBy(o => o.id).ToDictionary(
34              o => o.Key.ToString(),
35              r => r.Select(s => new {
36                  order_item_id = s.order_item_id.ToString(),
37                  product_id = s.product_id.ToString(),
38                  seller_id = s.seller_id.ToString(),
39                  shipping_limit_date = s.shipping_limit_date,
40                  price = s.price,
41                  freight_value = s.freight_value
42              }));
43
```

```
            Console.WriteLine($"Total amount: {groupList.Count}");
            int count = 0;
            foreach (var list in groupList)
            {
                var filter = Builders<BsonDocument>.Filter.Eq("
    order_id", list.Key);
                var update = Builders<BsonDocument>.Update.Set("
    order_items", list.Value);
                ordersCollection.UpdateOne(filter, update);
                Console.WriteLine($"{++count} completed");
            }

            Console.WriteLine("Press RETURN to exit");
            Console.ReadLine();
        }
```

3 MongoDB - Product search

3.1 Search queries

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.model.Filters;
import org.bson.Document;
import org.bson.types.ObjectId;
import com.mongodb.client.model.Aggregates;
import com.mongodb.client.model.Accumulators;
import java.util.List;
import com.mongodb.Block;

try {
  MongoCollection<Document> collection = vars.getObject("
    olist_products_dataset");
  //Reading 100 records
  List<Document> foodResults = collection.find(eq("
    product_category_name", "food")).toList();
  //Reading 1,000 records
  List<Document> musicResults = collection.find(eq("
    product_category_name", "music")).toList();
  //Reading 10,000 records
  List<Document> bedResults = collection.find(eq("
    product_category_name", "bed_bath_table")).toList();
  //Reading 100,000 records
  List<Document> toysResults = collection.find(eq("
    product_category_name", "toys")).toList();
}
catch (Exception e) {
  SampleResult.setSuccessful(false);
  SampleResult.setResponseCode("500");
  SampleResult.setResponseMessage("Exception: " + e);
}
```

3.2 Aggregate queries

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.model.Filters;
import org.bson.Document;
```

```
4 import org.bson.types.ObjectId;
5 import com.mongodb.client.model.Aggregates;
6 import com.mongodb.client.model.Accumulators;
7 import java.util.List;
8 import com.mongodb.Block;
9
10 try {
11   MongoCollection<Document> collection = vars.getObject("
     olist_products_dataset");
12   //Counting 100 records
13   collection.aggregate(Arrays.asList(
14         Aggregates.match(Filters.eq("product_category_name", "food"
     )),
15       Aggregates.group("product_category_name", Accumulators.sum("
     count", 1))
16   ));
17   //Counting 1,000 records
18   collection.aggregate(Arrays.asList(
19         Aggregates.match(Filters.eq("product_category_name", "music
     ")),
20       Aggregates.group("product_category_name", Accumulators.sum("
     count", 1))
21   ));
22   //Counting 10,000 records
23   collection.aggregate(Arrays.asList(
24         Aggregates.match(Filters.eq("product_category_name", "
     bed_bath_table")),
25       Aggregates.group("product_category_name", Accumulators.sum("
     count", 1))
26   ));
27   //Counting 100,000 records
28   collection.aggregate(Arrays.asList(
29         Aggregates.match(Filters.eq("product_category_name", "toys"
     )),
30       Aggregates.group("product_category_name", Accumulators.sum("
     count", 1))
31   ));
32 }
33 catch (Exception e) {
34   SampleResult.setSuccessful(false);
35   SampleResult.setResponseCode("500");
36   SampleResult.setResponseMessage("Exception: " + e);
37 }
```

3.3 Scalability testing

```
1 import com.mongodb.client.MongoCollection;
2 import com.mongodb.client.model.Filters;
3 import org.bson.Document;
4 import org.bson.types.ObjectId;
5 import com.mongodb.client.model.Aggregates;
6 import com.mongodb.client.model.Accumulators;
7 import java.util.List;
8 import com.mongodb.Block;
9 try {
10   MongoCollection<Document> collection = vars.getObject("
     olist_products_dataset");
```

```
    List<Document> results = collection.find(eq("product_id", "629
      beb8e-7317-703d-cc5f-35b5463fd20e")).toList();
    return results;
}
catch (Exception e) {
    SampleResult.setSuccessful(false);
    SampleResult.setResponseCode("500");
    SampleResult.setResponseMessage("Exception: " + e);
}
```

4 MongoDB - Order placement

4.1 Inserting orders

```
import com.mongodb.client.MongoCollection;
import org.bson.Document;
import java.util.Arrays;
import org.bson.types.ObjectId;

try {
    MongoCollection<Document> orders = vars.getObject("
    olist_orders_dataset");

    //Inserting 100 records
  List<Document> order100Items = new ArrayList<Document>();
  for (int i=0; i<100; i++) {
    Document item = new Document("order_item_id", UUID.randomUUID()
    .toString())
      .append("product_id", "4244733e-06e7-ecb4-970a-6e2683c13e61")
      .append("seller_id", "48436dad-e18a-c8b2-bce0-89ec2a041202")
      .append("shipping_limit_date", new Date())
      .append("price", "58.90")
      .append("freight_value", "13.29")
    order100Items.add(item);
  }
  Document newOrder100 = new Document("order_id", UUID.randomUUID()
    .toString())
      .append("order_status", "created")
      .append("customer_id", "cc643091-8875-1727-fe40-3d8570630495"
    )
      .append("order_purchase_timestamp", new Date())
      .append("order_items", order100Items);
  orders.insertOne(newOrder100);

  //Inserting 1,000 records
  List<Document> order1000Items = new ArrayList<Document>();
  for (int i=0; i<1000; i++) {
    Document item = new Document("order_item_id", UUID.randomUUID()
    .toString())
      .append("product_id", "4244733e-06e7-ecb4-970a-6e2683c13e61")
      .append("seller_id", "48436dad-e18a-c8b2-bce0-89ec2a041202")
      .append("shipping_limit_date", new Date())
      .append("price", "58.90")
      .append("freight_value", "13.29")
    order1000Items.add(item);
  }
```

```
38    Document newOrder1000 = new Document("order_id", UUID.randomUUID
        ().toString())
39          .append("order_status", "created")
40          .append("customer_id", "cc643091-8875-1727-fe40-3d8570630495"
        )
41          .append("order_purchase_timestamp", new Date())
42          .append("order_items", order1000Items);
43    orders.insertOne(newOrder1000);
44
45    //Inserting 10,000 records
46    List<Document> order10000Items = new ArrayList<Document>();
47    for (int i=0; i<10000; i++) {
48      Document item = new Document("order_item_id", UUID.randomUUID()
        .toString())
49          .append("product_id", "4244733e-06e7-ecb4-970a-6e2683c13e61")
50          .append("seller_id", "48436dad-e18a-c8b2-bce0-89ec2a041202")
51          .append("shipping_limit_date", new Date())
52          .append("price", "58.90")
53          .append("freight_value", "13.29")
54      order10000Items.add(item);
55    }
56    Document newOrder10000 = new Document("order_id", UUID.randomUUID
        ().toString())
57          .append("order_status", "created")
58          .append("customer_id", "cc643091-8875-1727-fe40-3d8570630495"
        )
59          .append("order_purchase_timestamp", new Date())
60          .append("order_items", order10000Items);
61    orders.insertOne(newOrder10000);
62  }
63  catch (Exception e) {
64    SampleResult.setSuccessful(false);
65    SampleResult.setResponseCode("500");
66    SampleResult.setResponseMessage("Exception: " + e);
67  }
```

4.2 Scalability testing

```
1   import com.mongodb.client.MongoCollection;
2   import org.bson.Document;
3   import java.util.Arrays;
4   import org.bson.types.ObjectId;
5
6   try {
7     MongoCollection<Document> orders = vars.getObject("
        olist_orders_dataset");
8     Document newOrder = new Document("order_id", UUID.randomUUID().
        toString())
9           .append("order_status", "created")
10          .append("customer_id", "cc643091-8875-1727-fe40-3d8570630495"
        )
11          .append("order_purchase_timestamp", new Date())
12    orders.insertOne(newOrder);
13  }
14  catch (Exception e) {
15    SampleResult.setSuccessful(false);
16    SampleResult.setResponseCode("500");
```

```
17   SampleResult.setResponseMessage("Exception: " + e);
18 }
```

5 MongoDB - Order update

5.1 Updating orders

```
1 import com.mongodb.client.MongoCollection;
2 import static com.mongodb.client.model.Updates.*;
3 import static com.mongodb.client.model.Filters.*;
4 import org.bson.Document;
5 import java.util.Arrays;
6 import org.bson.types.ObjectId;
7
8 try {
9   MongoCollection<Document> collection = vars.getObject("
      olist_orders_dataset");
10
11   //Updating 100 records
12   collection.updateMany(eq("customer_id", "06b8999e-2fba-1a1f-bc88
      -172c00ba8bc7"), combine(set("order_status","approved"), set("
      order_approved_at",new Date())));
13
14   //Updating 1,000 records
15   collection.updateMany(eq("customer_id", "47c42b50-ab59-928b-531c
      -450e672ce223"), combine(set("order_status","approved"), set("
      order_approved_at",new Date())));
16
17   //Updating 10,000 records
18   collection.updateMany(eq("customer_id", "f6cc975f-9ed5-f7d6-d78d
      -5f8ac711a7ac"), combine(set("order_status","approved"), set("
      order_approved_at",new Date())));
19 }
20 catch (Exception e) {
21   SampleResult.setSuccessful(false);
22   SampleResult.setResponseCode("500");
23   SampleResult.setResponseMessage("Exception: " + e);
24 }
```

5.2 Scalability testing

```
1 import com.mongodb.client.MongoCollection;
2 import static com.mongodb.client.model.Updates.*;
3 import static com.mongodb.client.model.Filters.*;
4 import org.bson.Document;
5 import java.util.Arrays;
6 import org.bson.types.ObjectId;
7
8 try {
9   MongoCollection<Document> collection = vars.getObject("
      olist_orders_dataset");
10   collection.updateOne(eq("order_id", "4eeecb9e-6575-2502-4b9b
      -538730c5f974"), combine(set("order_status","approved"), set("
      order_approved_at",new Date())));
11 }
12 catch (Exception e) {
```

```
    SampleResult.setSuccessful(false);
    SampleResult.setResponseCode("500");
    SampleResult.setResponseMessage("Exception: " + e);
}
```

6 MongoDB - Order deletion

6.1 Deleting orders

```
import com.mongodb.client.MongoCollection;
import static com.mongodb.client.model.Updates.*;
import static com.mongodb.client.model.Filters.*;
import org.bson.Document;
import java.util.Arrays;
import org.bson.types.ObjectId;

try {
    MongoCollection<Document> collection = vars.getObject("
        olist_orders_dataset");

    //Deleting 100 records
    collection.deleteMany(eq("customer_id", "06b8999e-2fba-1a1f-bc88
        -172c00ba8bc7"));

    //Deleting 1,000 records
    collection.deleteMany(eq("customer_id", "47c42b50-ab59-928b-531c
        -450e672ce223"));

    //Deleting 10,000 records
    collection.deleteMany(eq("customer_id", "f6cc975f-9ed5-f7d6-d78d
        -5f8ac711a7ac"));
}
catch (Exception e) {
    SampleResult.setSuccessful(false);
    SampleResult.setResponseCode("500");
    SampleResult.setResponseMessage("Exception: " + e);
}
```

6.2 Scalability testing

```
import com.mongodb.client.MongoCollection;
import static com.mongodb.client.model.Updates.*;
import static com.mongodb.client.model.Filters.*;
import org.bson.Document;
import java.util.Arrays;
import org.bson.types.ObjectId;

try {
    MongoCollection<Document> collection = vars.getObject("
        olist_orders_dataset");
    collection.deleteOne(eq("order_id", "e3447938-231f-b6ba-2fee-0231
        b51eca59"));
}
catch (Exception e) {
    SampleResult.setSuccessful(false);
    SampleResult.setResponseCode("500");
```

```
15    SampleResult.setResponseMessage("Exception: " + e);
16 }
```

7 PostgreSQL JDBC connection

Connect JMeter to PostgreSQL with JDBC configuration

```
1 Database URL: jdbc:postgresql://localhost:5432/brazilian_ecommerce?
      autoReconnect=true
2 JDBC Driver class: org.postgresql.Driver
3 Username: xxxxxx
4 Password: xxxxxx
```

8 PostgreSQL - Product search

8.1 Search queries

```
1 -- Reading 100 records
2 select * from olist_products_dataset where product_category_name =
      'food'
3
4 -- Reading 1,000 records
5 select * from olist_products_dataset where product_category_name =
      'music'
6
7 -- Reading 10,000 records
8 select * from olist_products_dataset where product_category_name =
      'bed_bath_table'
9
10 -- Reading 100,000 records
11 select * from olist_products_dataset where product_category_name =
      'toys'
```

8.2 Aggregate queries

```
1 -- Counting 100 records
2 select product_category_name, count(product_category_name) as
      number from olist_products_dataset where product_category_name=
      'food' group by product_category_name
3
4 -- Counting 1,000 records
5 select product_category_name, count(product_category_name) as
      number from olist_products_dataset where product_category_name=
      'music' group by product_category_name
6
7 -- Counting 10,000 records
8 select product_category_name, count(product_category_name) as
      number from olist_products_dataset where product_category_name=
      'bed_bath_table' group by product_category_name
9
10 -- Counting 100,000 records
11 select product_category_name, count(product_category_name) as
      number from olist_products_dataset where product_category_name=
      'toys' group by product_category_name
```

8.3 Scalability testing

```
select * from olist_products_dataset where product_id = '629beb8e
    -7317-703d-cc5f-35b5463fd20e';
```

9 PostgreSQL - Order placement

9.1 Inserting orders

```
-- Inserting 100 records
INSERT INTO olist_order_items_dataset(order_id,product_id,
    seller_id, shipping_limit_date, price, freight_value)
(SELECT 'f7f82cfc-1e18-47e8-8426-13a79a82c32d','4244733e-06e7-ecb4
    -970a-6e2683c13e61','48436dad-e18a-c8b2-bce0-89ec2a041202','${
    __time(yyyy-MM-dd'T'hh:mm:ssZ)}',58.90,13.29 FROM
    generate_series(1, 100) as x)

-- Inserting 1,000 records
INSERT INTO olist_order_items_dataset(order_id,product_id,
    seller_id, shipping_limit_date, price, freight_value)
(SELECT 'f7f82cfc-1e18-47e8-8426-13a79a82c32d','4244733e-06e7-ecb4
    -970a-6e2683c13e61','48436dad-e18a-c8b2-bce0-89ec2a041202','${
    __time(yyyy-MM-dd'T'hh:mm:ssZ)}',58.90,13.29 FROM
    generate_series(1, 1000) as x)

-- Inserting 10,000 records
INSERT INTO olist_order_items_dataset(order_id,product_id,
    seller_id, shipping_limit_date, price, freight_value)
(SELECT 'f7f82cfc-1e18-47e8-8426-13a79a82c32d','4244733e-06e7-ecb4
    -970a-6e2683c13e61','48436dad-e18a-c8b2-bce0-89ec2a041202','${
    __time(yyyy-MM-dd'T'hh:mm:ssZ)}',58.90,13.29 FROM
    generate_series(1, 10000) as x)
```

9.2 Scalability testing

```
INSERT INTO olist_orders_dataset(order_status, customer_id,
    order_purchase_timestamp) VALUES ('created','cc643091
    -8875-1727-fe40-3d8570630495','${__time(yyyy-MM-dd'T'hh:mm:ssZ)
    }');
```

10 PostgreSQL - Order update

10.1 Updating orders

```
-- Updating 100 records
update olist_orders_dataset od set order_status='approved',
    order_approved_at='${__time(yyyy-MM-dd'T'hh:mm:ssZ)}' from (
    select * from olist_orders_dataset where customer_id='06b8999e
    -2fba-1a1f-bc88-172c00ba8bc7') as fo where od.order_id = fo.
    order_id

-- Updating 1,000 records
```

```sql
update olist_orders_dataset od set order_status='approved', order_approved_at='${__time(yyyy-MM-dd'T'hh:mm:ssZ)}' from (select * from olist_orders_dataset where customer_id='47c42b50-ab59-928b-531c-450e672ce223') as fo where od.order_id = fo.order_id

-- Updating 10,000 records
update olist_orders_dataset od set order_status='approved', order_approved_at='${__time(yyyy-MM-dd'T'hh:mm:ssZ)}' from (select * from olist_orders_dataset where customer_id='f6cc975f-9ed5-f7d6-d78d-5f8ac711a7ac') as fo where od.order_id = fo.order_id
```

10.2 Scalability testing

```sql
update olist_orders_dataset set order_status='approved', order_approved_at='${__time(yyyy-MM-dd'T'hh:mm:ssZ)}' where order_id='4eeecb9e-6575-2502-4b9b-538730c5f974'
```

11 PostgreSQL - Order deletion

11.1 Deleting orders

```sql
-- Deleting 100 records
delete from olist_orders_dataset where customer_id='b8999e -2fba -1a1f -bc88-172 c00ba8bc7'

-- Deleting 1,000 records
delete from olist_orders_dataset where customer_id='c42b50 -ab59-928b-531c-450 e672ce223'

-- Deleting 10,000 records
delete from olist_orders_dataset where customer_id='cc975f -9ed5 -f7d6 -d78d-5 f8ac711a7ac'
```

11.2 Scalability testing

```sql
delete from olist_orders_dataset where order_id='e3447938-231f-b6ba-2fee-0231b51eca59'
```