A novel Decision Support System for modern Relational and Non-Relational Database Systems

PW Jordaan

orcid.org/0000-0003-3022-2715

Thesis accepted in fulfilment of the requirements for the degree *Doctor of Philosophy in Engineering* with Computer and Electronic Engineering at the North-West University

Promoter: Prof JEW Holm

Graduation: June 2023
Student number: 20484127

Soli Deo Gloria

"For the glory of God alone"

Acknowledgements

I thank my Lord, Jesus Christ, for giving me the ability and the support structure to complete this thesis. No words can express the gratitude I have for being able to write this section.

To my family and friends who have carried me—sometimes literally—during difficult times, long hours, sacrifice, self-doubt and weariness, thank you!

To my beautiful wife, Jeanne-mari, you are an unyielding force of inspiration. You have enabled me to finish this study in more ways than I can convey. Thank you for your continual intercession and support and for believing in me.

To my children (all post-starting this endeavour), Annabel and Joshua, you bless me deeply every day.

To the best mother anyone could ask for: I miss you and know you wished dearly for me to finish this doctorate.

Prof Johann Holm, you have been both an anchor and a rudder. If I may exploit that analogy, you have kept me from drifting off-topic and steered me in the right direction where needed—in academics, career and life—. Thank you for all your hard work in reviewing and guiding this thesis!

Thank you to Dr Melvin Ferreira for all the support and aid with administrative tasks.

To my friends and colleagues at Jericho Systems, thank you for your unyielding support, motivation and infinite inspiration.

Abstract

Data-based application design and development can be complex and difficult to navigate. With relational and non-relational database technologies, each with divergent paradigms, it is not apparent how to make the correct technology and design choices. Cloud-based environments add additional complexity to the decision-making process. Traditional methodologies and paradigms often neglect development requirements that must be derived from higher-level operational and maintenance requirements. A need for a guide or strategy that can direct a designer according to a defined process became evident. Procedures, methods, and techniques were further needed to assist the designer in making complex choices. This thesis follows the Quality Research Management method in concert with the Design Science Research paradigm to synthesise a pragmatic Decision Support System in the form of a process model from literature, experimentation, subject matter expertise, and best practice methods. Empirical results showed that the performance difference between the competing database technologies evaluated in this research is not the determining factor. The correct design procedure, choices, and methods in the preliminary design phase are critical performance determinants. The artefact developed in this research is a database development process model that includes a process based on the systems engineering development process, with associated procedures and methods that support decision-making. The process model was applied to the well-known TPC-C test case to validate its effectiveness and its pragmatic value. The Decision Support System, with the novelty of being mostly technology agnostic, improved database performance by up to 20% relative to traditional methods on the TPC-C test case and provided evidence that it has effectively addressed the initial research problem.

Keywords: Database modelling; Systems paradigm; Holistic perspective

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

ACID	Atomicity Consistency Isolation Durability
BASE	Basic Availability, Soft state, Eventual consistency
BSON	Binary JavaScript Object Notation
CRUD	Create Read Update Delete
DBA	Database Administrator
DBMS	Database Management System
DSR	Design Science Research
DSS	Decision Support System
EAV	Entity Attribute Value
ER	Entity Relational
ERD	Entity Relationship Diagram
FK	Foreign Key
ICT	Information and Communications Technology
IoT	Internet of Things
JSON	JavaScript Object Notation
NoSQL	Not-only SQL
ORDBMS	Object Relational Database Management System
PK	Primary Key
QRM	Quality Research Management
RD	Routine Design
RDBMS	Relational Database Management System
RVM	Research Validation Matrix
SE	Systems Engineering
SQL	Structured Query Language
TPM	Technical Performance Measures
TpmC	Transactions per minute C
UML	Unified Modelling Language

There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success than to take the lead in the introduction of a new order of things.

Niccolo Machiavelli

Chapter 1

Introduction

1.1 Overview

This thesis presents the process, findings, and results of a research project on synthesising and evaluating a decision support system (DSS) for modelling using modern relational and non-relational database technologies. The debate on a best practice approach for database design is usually contained in a specific ontology encapsulated in a particular worldview. As a result, it is difficult to find an unbiased perspective on database design and the design process is often constrained by the choice of technology.

In order to find an unbiased perspective, it is necessary to consider different world views, as discussed in this thesis. Furthermore, it is prudent to "stand back" and consider a database's development (as opposed to design) using a recognised multi-disciplinary development method as a reference. In doing so, the function and performance of a data-based system take precedence over form, resulting in a system tailored for an application, as opposed to a solution being forced into an application.

The challenge to be addressed is to find a suitable method for database development, based on more than technical database design methods, that is technology agnostic to some extent. It is acknowledged that there will always be some dependency on technology — a fact that cannot be ignored. Nonetheless, the research effort is to establish which technology will perform best when essential database functions are performed. This will assist in choosing technology when the time arises and contributes to the novelty of the DSS.

A robust development process must be followed in any given development project, including specific design tasks. The challenge is thus to also find a generic process, in context, that will guide a developer in an environment that is broader than just an information technology space. Such a process is found in systems engineering, with a specific focus on the life cycle of information technology systems. A development procedure is defined in the most appropriate life cycle phase to provide developers with clear guidance.

Research validation had to be provided at the end of this research project, given a proposed process, methods, and techniques (the "schema"). This was done using a test case that is representative of a real-world problem. A pragmatic guide (DSS) for database design resulted from a comprehensive literature study, empirical experimentation, process model design, and test case evidence. Background to the research challenge is provided in the sections that follow.

1.2 Background

Before the research project is presented, some background is given to provide context to the reader. This study takes place in an information technology environment; more specifically, a cloud environment is targeted, which has a unique set of challenges to address [1]:

Data consistency Ensuring all data elements are in a consistent state, regardless of infrastructure or other failures;

Availability The ability to perform under varying conditions while preventing loss of service;

Predictable performance Shared cloud resources have turbulent performance and a stable, predictable solution is desirable;

Scalable and high-performance storage The ability to dynamically expand storage (and processing power) without downtime.

The DSS will be based on empirical data from experimentation on database performance behaviour and best practices for data modelling across a broad range of applications. Due to the intricate relationship between a data-based application and database modelling, the proposed strategy will focus on the full life cycle of a system, taking application query patterns into account in the process.

Data models are often represented by *Unified Modelling Language (UML)* diagrams which consist of *classes*, *attributes*, *methods*[1] and *relationships*. Instances of *classes* are called *objects*. Figure 1.1 shows a typical data model as a UML class diagram [2].

Typical relationships in UML diagrams are *inheritance* and *association* (including the stronger *aggregation*). *Inheritance* refers to classes with inherited attributes or methods from a super/parent class. An example of *inheritance* could be employees and administrators, where administrators are, in fact, also employees. One can therefore say administrators *inherit* the attributes and methods of employees. *Associative* relationships have a *cardinality* or *multiplicity* attribute which signifies the minimum and the maximum number of associated objects in the relationship [2].

[1]Methods are traditionally not considered during database design but during the application design

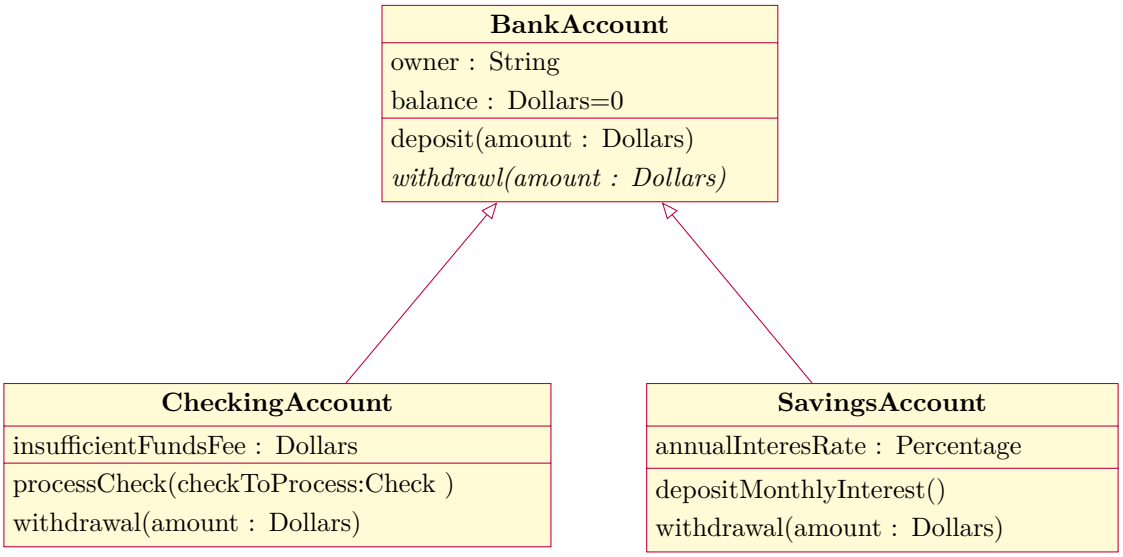


Figure 1.1: UML Diagram

Relational database management systems (RDBMSs) and, more specifically, the relational data model was introduced as early as the 1970s. Since then, many enhancements have been made, and RDBMSs have become the most popular database systems in the information technology sector [3].

RDBMSs traditionally employ a data model consisting of *tables*, *rows and columns* and *relationships between tables*. Relationships between tables are represented by associating a primary key (PK), a unique data field, of one table with a foreign key (FK) in another table. The process of translating a UML data model to the RDBMS domain, following the typical paradigm, is summarized in Table 1.1 by showing the equivalent concepts in both. The modelling process will be elaborated on in Chapter 3 [2].

Table 1.1: UML to RDBMS conversion

UML	RDBMS
Attributes	Columns
Classes	Tables
Objects	Rows
1-Many Association	PK → FK
Many-Many Association	New intermediate table with two 1-Many associations linking to the two tables and optional discriminator
1-1 Association[2]	PK → FK
Inheritance	± 1-1 association (PK → FK)

Relational databases have transactional capability, implying the ability to manipulate data in a multi-user environment while preventing uncontrolled and undesired interactions. This concept has been more popularly coined in the term ACID compliance, which is an

[2]This is a special case of a 1-Many association

acronym described below [4]:

Atomicity
Transactions are all-or-nothing, and the state of the data is always known to the user;

Consistency
After completion of the transaction, the database will be consistent in that it has committed the results legally and predictably;

Isolation
Each event executed within a transaction must be hidden from concurrent transactions;

Durability
The ability to ensure a transaction was successfully applied and persisted.

It is this ACID compliance that makes RDBMSs popular, effective and trusted. Typical database operations include *create*, *read*, *update* and *delete*. These operations are often referred to as CRUD operations and form the basis of database applications. In this thesis PostgreSQL[3] will be used as the RDBMS for research [3].

A new type of DBMS called NoSQL or Not-only SQL databases was introduced in 1998 but became more popular only recently (2009). These NoSQL databases were designed to offer high performance and availability by trading full ACID compliance for some BASE (Basic Availability, Soft state, Eventual consistency) attributes. As there are more than 120 different NoSQL databases, this thesis will study a document store called MongoDB[4] [5].

MongoDB is not relational in the traditional sense but is rather document-oriented in design. A MongoDB database exists out of *collections of documents*, where each document can have any number of sub-documents and fields. A typical MongoDB document is shown in JSON (JavaScript Object Notation) format in Listing 1.1, although the database stores documents as a binary variation called BSON. This model shows that documents (comments in this case) can be embedded within other documents and support for arrays of fields are present. This flexibility in document-oriented data models makes it a powerful companion or substitute for RDBMSs in many use cases [6, 7].

Table 1.2 shows the correlation between UML models and document-oriented data models, which will be elaborated on in Chapter 3. It should be noted that due to MongoDB's flexible and schemaless nature, many representations could be made.

[3]PostgreSQL is the most popular open-source ORDBMS according to `http://db-engines.com/en/ranking` in September 2021

[4]MongoDB is the most popular open-source NoSQL database according to `http://db-engines.com/en/ranking` in September 2021

```
{
  "_id": "5b273446aa567a2ee930235",
  "title": "A new blog post",
  "content": "...",
  "comments": [
    {
      "name": "John",
      "email": "john@example.com",
      "content": "interesting"
    },
    {
      "name": "George",
      "email": "george@example.com",
      "content": "excellent post!"
    }
  ]
}
```

Listing 1.1: JavaScript Object Notation

Table 1.2: UML to MongoDB document conversion

UML	MongoDB
Attributes	Fields in a document
Classes	Collections
Objects	Documents
1-Many Association	Either an array of references to another collection's documents or an array of embedded sub-documents
Many-Many Association	A join collection (analogous to an RDBMS' intermediate table), or fully embedded[5], or embedding of document references
1-1 Association[6]	Either a reference to another collection's document or embedded documents
Inheritance	Embedded documents or inherited fields

It is evident that UML data models have valid representations for both relational and non-relational data models. [8]

A study by Zhao et al. demonstrating that MongoDB can perform relational algebra (the building blocks of relational databases) on par with relational databases further reinforces the notion that either can be used to build applications functionally [9].

Now the question arises: RDBMS or NoSQL? This question usually cannot be answered without some form of bias, prejudice or experience. Specific scenarios should exist where one or the other will be the better choice, but as a complete solution, either require some trade-off. This is even more prevalent when an application involves scalability, flexibility and high availability when deployed to a cloud computing environment [6, 10–12].

1.3 Rationale and Related Work

At the time of this research, no definitive solution or design guidelines were found that incorporate modelling of both relational *and* non-relational database types (or a hybrid) in a cloud-based context. Hybrid approaches are problem and project-specific, and no known strategy for combining the two data model paradigms generically or procedurally could be found in the literature.

Furthermore, no DSS was found that guides the design in a heterogenous form while remaining technology agnostic through the concept and preliminary design phases, nor incorporating any Systems Engineering (SE) concepts.

IBM has developed a proprietary system, *Informatix*, allowing both MongoDB and SQL actions. Still, it does not solve the problem of where to store specific data entities, nor does it provide strategies for effectively utilizing either MongoDB or SQL. *Informatix* only allows access to both databases with traditional SQL *and* MongoDB connectors [13].

Similarly, middleware translation software has been developed to try and bridge the gap by allowing SQL queries to NoSQL databases. This feature makes it convenient to work with by using only one query language, but it does not help during the design phase of such a hybrid system. It only considers duplication of all data and does not allow for mixing strategies [14].

Another middleware solution uses a data dictionary to catalogue specific data collections to a NoSQL or a relational database. It also allows SQL queries to both database types [15].

Attempts to give NoSQL databases ACID properties (transactional support) have also been made, but they still don't provide a complete hybrid between the two paradigms

[5] This is limited to a total document size of 16MiB
[6] This is a special case of a 1-Many association

[16].

MongoDB has launched support for transactions across shards in version 4.2, which addresses the ACID/transactional requirement for many applications. However, scenarios, where SQL or table-based access is still a requirement, remain unanswered [17].

Kanade and Gopal have proposed a hybrid data model for MongoDB and evaluated the performance based on the level of normalization and embedding. The results given show that the data model greatly influenced the performance. It also shows that a semi-normalized MongoDB design can be compared with a normalized RDBMS system. The authors suggested further studying a strategy for data modelling and investigating a platform to utilize both types of databases [18].

An attempt was made to model a hybrid database in ERD (entity relationship diagram) by Villari et al.. They added additional markup annotations to ERD to show where big-data interaction occurs. However, they do not provide a strategy or framework for designing the logical or physical databases nor describe the application-level requirements [19].

According to Badia and Lemire, it is important to use good database design methodologies to have consistent databases that remain mostly unchanged. They claim that application and data design are uncoupled, and make the following submission to the research world, asking for further research on the following [20]:

1. Design for a distributed world;
    - 1.1 Design methodologies must be updated (including NoSQL);
    - 1.2 Easier data integration;
2. Rethink functional dependencies in schemas;
    - 2.1 Normalization;
    - 2.2 Replacement for functional dependence;
    - 2.3 Modelling functional dependencies;
    - 2.4 Physical design vs. logical design;
3. Design for imperfect knowledge;
    - 3.1 Growing scope;
    - 3.2 Permissiveness of data;
    - 3.3 Open-world model;
    - 3.4 NoSQL clarification;
    - 3.5 Probabilistic databases;

The above call for research validates the research problem of this research project and confirms that, at the time, effort was required to address their request.

A recent study (2018) by Imam et al. appeals to NoSQL academia to help guide the database design process, especially for NoSQL environments. They provide guidelines to aid the database developer in designing scalable database models. This study, however, focuses only on document-oriented databases and does not consider relational databases with a modern paradigm. Furthermore, the guidelines are rigid and do not specifically guide a developer through a design process, nor does it consider the application interfaces [21].

Imam et al. developed a proposition model for automating schema generation by inputting entities and relationships and create, read, update and delete operations. They found that the generated schema performs better than a naive traditional approach [22].

1.4 Proposal

This thesis will synthesise a strategy to effectively produce a hybrid relational (normalised) and non-relational (denormalised) data model in data-based applications. By modelling a generic model in UML and translating it to relational *and* non-relational data models, a hybrid data model can be created using a "best-of-breed" approach.

Identifying strengths and shortcomings in both data models can provide a means of selecting the most desirable traits between both models for a specific use case or specific data object.

Notably, purely NoSQL or RDBMS as a solution is a special case of the hybrid approach wherein 100% of the resources and modelling are assigned to only one of the technologies. The principles applied, however, in making the design choices will remain the same. Similarly, when considering the underlying data modelling techniques used, regardless of the technology, trade-offs between purely normalised or purely denormalised have the same outcome.

A holistic view of the system is observed by employing a systems engineering approach instead of the traditional middle-out ER approach. Functional units, their interfaces and the database system are designed *together*, with a full life cycle view, incorporating all related functional units of the system. A holistic view is in contrast to a separated database-only and application-only design approach typically followed today and will lead to more scalable and flexible designs.

To rationalise the above statements, consider the IDEF0 function diagram in Figure 1.2. A functional unit in a system has an input, output, controls or constraints, and mechanisms or resources weighing in on the function's design, modelling and analysis. Input in this context is the system requirements. Resources include human resources, facilities, maintenance and support. Controls and constraints can include technical or economic constraints. All controls must be considered to successfully design a function, including those that may come from preliminary design steps and other functional units [23].

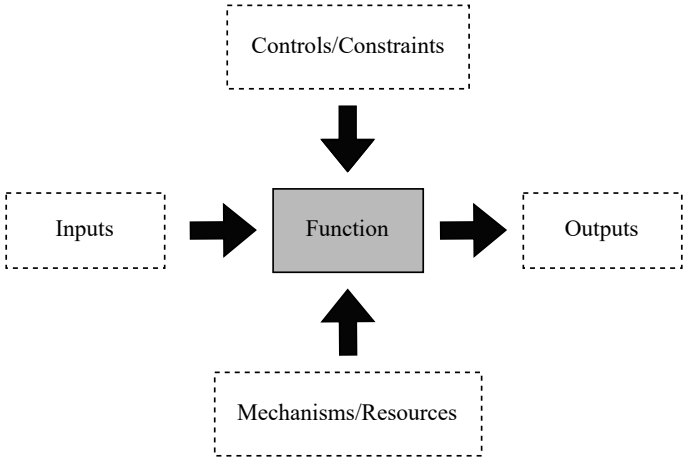


Figure 1.2: Function Diagram

Moreover, a system consists of many such functions interconnected with interfaces that must be considered during the design, further advancing the need for a holistic database modelling and design paradigm [12, 20, 21].

This thesis submits that by employing systems engineering (SE) principles with a holistic view of the system, the problems mentioned above become manageable and will produce scalable and flexible systems. In contrast to ER, SE follows a top-down design approach [23–25].

1.4.1 Thesis Outline

The thesis follows the structure as outlined below:

Chapter 2

This chapter presents the research methodology *and* problem statement. These are combined as the problem statement provides context to the research methodology. Different world views are considered, the research paradigm, methodology, and management processes are defined, and the problem is defined in terms of individual research challenges.

Chapter 3

A comprehensive literature study on databases, application development, and systems engineering is provided. Two database paradigms are considered, and findings are made on a hybrid approach where both relational and non-relational databases are employed. The systems engineering process is analysed, and the differences between focused information technology development and multi-disciplinary system development are highlighted.

Chapter 4

A performance comparison between the two database technologies (DBMSs) is presented. Experiments are described with empirical results that show how the two

technologies performed on foundational database functions by varying design parameters. These results are important as they are often used to promote a specific technology (paradigm) at the cost of generalisation.

Chapter 5

A decision support system (DSS), obtained by considering the phases of Systems Engineering design process, best practice design methods and techniques, results from Chapter 4, and pragmatic consideration, is proposed. The guide is not exhaustive but pragmatic and provides the developer with a process model comprising decision guidelines and critical considerations.

Chapter 6

The DSS is applied to a test case and demonstrates the value of following a technology-agnostic design process. The resulting data model provided evidence of the DSS's effectiveness and confirmed that a modern approach can improve performance.

Chapter 7

Finally, the last chapter provides a summary of the research process, findings, and results; it concludes with validation of the research as a whole.

No problem can be solved from the same level of consciousness that created it.

Albert Einstein

Chapter 2

Problem Statement & Research Methodology

Introduction

A directed research environment requires project management and a research paradigm to ensure a quality process is followed and research outcomes address real-world problems. As a result, this research applies Quality Research Management (QRM) inside the Design Science Research (DSR) paradigm. The QRM process allows systematic management and traceability in the research project [26], while DSR translates between real-world challenges and challenges in the academic domain [27].

This chapter builds on the introduction and background provided in the previous chapter by defining the problem in real-world and research terms.

After the problem has been stated, the research paradigm, methodology, and processes followed by this research project are clarified. Furthermore, expected evidence and contributions are described.

2.1 Philosophy

Given the context above, a pragmatic research philosophy had to be followed to obtain a grounded and practical solution [28].

Different views and predispositions exist in the database world when considering the proper design and modelling of data and how it can be integrated into a system. Moreover, more prejudice and misconceptions appear when different technologies are added to the equation. When the full life cycle of a system is considered, additional engineering and design considerations affect design choices. Is there an absolute best technology or design

methodology, or could there be more than one usable choice depending on the context, constraints or resources?

When considering literature and practical experience, there are contradicting truth statements, whether from biased or variable contexts. Therefore, the above line of thinking defines the context for the ontology adhered to by this thesis — there are multiple realities in the database and ICT (Information and Communications Technology) world competing for truth. Furthermore, Systems Engineering brings a different perspective by considering the impacts on the database design process across the full life cycle of a system. The ontology is summarized in Figure 2.1.

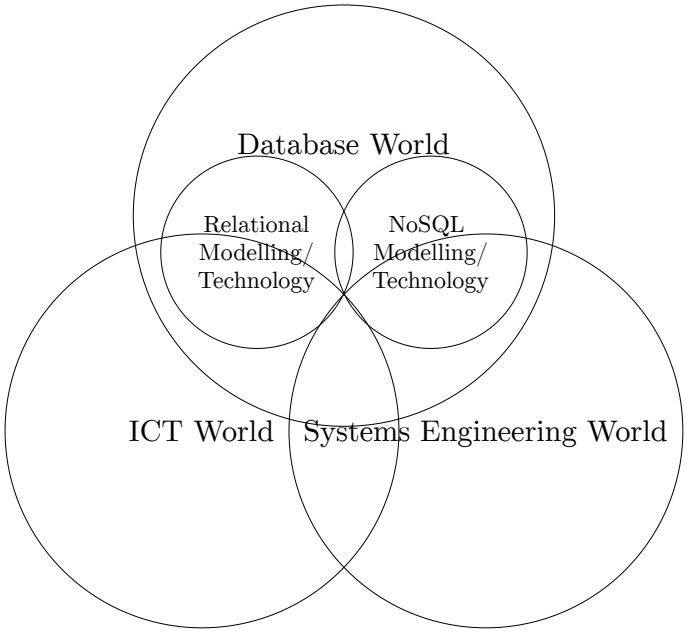


Figure 2.1: Ontology Representation

The Database World forms a part of the ICT World, with the ICT World forming part of the Systems Engineering World. These different perspectives sometimes conflict with ICT projects as they compete for ownership. Decisions made in each world will affect design choices and outcomes, which will significantly impact the final product's function, form, and fit, as well as its ultimate performance in practice. It is, therefore, important to consider this context in the research conducted here.

Considering the multiple possible truths and their individual contexts (regarding technology, design choice and methodologies), it is important to establish the contexts for which each is true. The epistemological requirement for the research philosophy is thus one where existing views and knowledge should be challenged to ensure unbiased, informative results that should lead to a pragmatic conclusion. It thus becomes evident that a single worldview will not suffice and that the above views and their interrelations must be considered when the research philosophy is defined.

2.2 Real-world Problem Context

For directed research, a real-world problem must be defined. The real-world problem is defined as follows:

How does a real-world data-based application benefit from pragmatic design and implementation?

As mentioned in the previous chapter, it is increasingly difficult to design data-based applications to adhere to modern requirements, especially given the vast array of options in database technology and modelling techniques. Often, puristic views forfeit proper systematic and pragmatic approaches to design and implementation, leading to problems in scalability and flexibility.

2.3 Research Management Framework

This research was conducted in the Quality Research Management Framework (QRM).

Other methods were considered, but the DSR paradigm was found to be applicable as the output of this research includes an artefact in the form of a design guideline or decision support system. QRM is a management framework that was designed to support the management of design research as it incorporates the following:

- DSR is a paradigm, with QRM as a framework and method to simplify the management of design science research. There is no clearly documented alternative to such a management framework as it provides the following [26]:
  - QRM was specifically designed to support DSR as DSR is simply a paradigm and not a methodology;
  - Traceability from real-world problem to research solutions in a manner that supports understanding;
  - Validation and verification are simplified concerning the different phases of research (problem analysis, literature, and solution);
- This research was defined in the "pragmatic" domain, which required a method that converted real-world challenges to workable solutions by following a systematic process;
- Further to the DSR paradigm and QRM framework (management), the actual methods included documentary research in the literature study and experimental research.

The QRM framework defines specific phases present in the research process. These allow the researcher to focus the research effort as follows [26]:

Inception/definition phase and baseline

A single need is addressed as a research goal. This is expanded into research challenges.

Analysis phase and baseline

A systematic breakdown of the research challenges is done and leads to a focused literature study, to better understand the challenges and identify possible high-level solutions.

Synthesis phase and baseline

Concept solutions are iteratively synthesised into specific solutions using the literature study results and creativity as inputs. In DSR artefacts are typically produced as specific solutions in the form of constructs, methods, models or instantiations.

Validation phase and baseline

Each phase is documented in a research validation matrix (RVM), which provides the ability to trace the validity of research from the final solution back to the original challenges.

Communication

The RVM serves as a communication of the research project. Not only does it visually show the research process, track progress and document results, but it also shows the validation in a traceable manner.

The framework is depicted in Figure 2.2. From the top-left real-world needs and the environment is fed into the research framework. Multiple overarching layers handle it: (i) the Systems Engineering Environment, (ii) Quality Research Management, (iii) Design Science Research, (iv) the processes, knowledge, resources and data available to solve the problem. The problem is processed and solved, following the same layers in reverse to the output. The output consists of artefacts that add to the body of knowledge concerning databases, ICT and engineering systems, specifically, in this case.

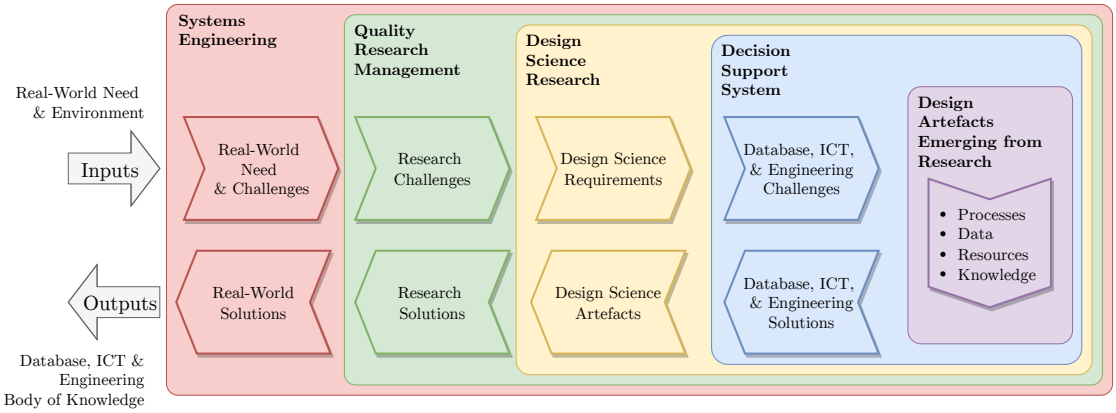


Figure 2.2: Research framework

The real-world problem is to design and implement a database from a systems perspective. This is a difficult real-world problem to which a solution was not evident at the onset of

this research (as evidenced by the recent Badia and Lemire publication in "A Call to Arms" in 2011).

No definitive framework/guideline was found in the literature at the level addressed in this thesis — focus was only on lower-level homogenous approaches. Therefore, a system-level DSS was required as a real-world need. From the need, requirements were derived in line with the DSR paradigm's relevance cycle as supported by the QRM framework.

2.4 Research Paradigm

This research project followed the DSR approach summarized in Figure 2.3. The real-world problem (in the form of use cases) is translated into an abstract and academic form through analysis. By designing a solution in the abstract world, an implementation or prototype is born and translated back into the real world as an artefact [2]. The research is then verified and validated during the translation from and to the real world [29].

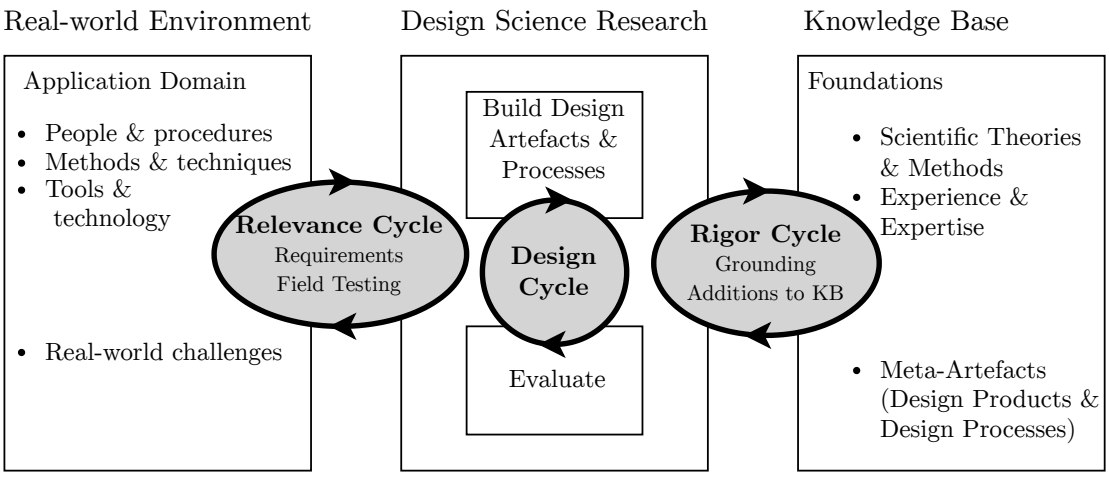


Figure 2.3: Design science research approach adapted from [30, Fig. 1]

It is important to prevent routine design (RD) from entering the DSR paradigm since it does not contribute to the scientific knowledge base. From the items in the left column of Table 2.1 it is clear what kind of contributions should be made - these form part of the research design criteria [29].

2.5 Research Problem

For research to be relevant and valid, a real-world problem must be translated into a theoretical and academic problem. This theoretical problem is also referred to as the research problem. The research problem of this thesis is defined as follows and discussed below:

Table 2.1: DSR vs. RD

Design Science	Routine Design
General solution	Specific solution
Novelty knowledge	Use of existing knowledge
Contribution to knowledge base	Known design (replication)
Solutions to new problems in new ways	Solve problems with existing knowledge
Technology invention	Technology application
Addresses abstract problems	Addresses specific problems
Solves a type of problem	Solves only one case

What are the critical database design and implementation factors that affect database performance?

According to Misra, pursuing performance by applying performability engineering also ensures a high-quality product which is reliable, maintainable and sustainable [31]. It is this concept of performance which leads to the research problem statement.

By identifying critical database design and implementation factors affecting database *performance*, the benefits directed by the real-world problem can be determined effectively in the research and be applied through a pragmatic approach.

Figure 2.4 shows the holistic systems framework adapted from a commonly used framework encountered in systems engineering texts [23–25]. This framework is used throughout this thesis to provide context for identifying critical factors and the contributions made from this research.

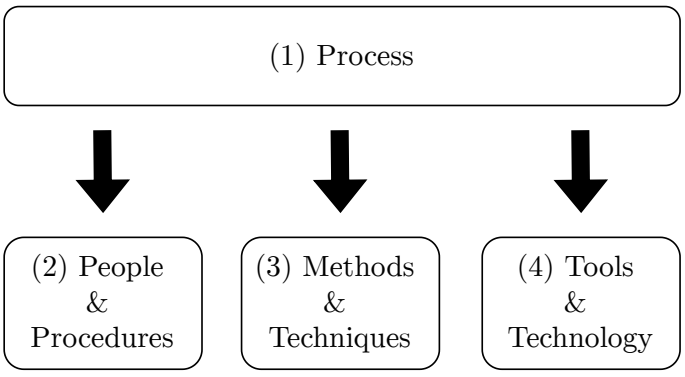


Figure 2.4: Holistic context based on a systems engineering framework [23]

To overcome the research problem, the following research challenges were identified and addressed throughout this research project by employing established research methods. For pragmatic solutions to emerge, it is important to view challenges systematically; they were therefore categorised into one of the context areas of Figure 2.4.

(1) Process

- *Full life cycle* approach not considered in methodologies;
- There is a need for a definitive guide or strategy for designing and planning data-based applications in the ICT full life cycle.

(2) People & Procedures

- Application and data model design are usually *separated*.

(3) Methods & Techniques

- *Conceptual modelling* approaches are almost exclusively for relational models;
- No equivalent *logical modelling* approach for both data model paradigms;
- Design and implementation *criteria* for each model type are unknown.

(4) Tools & Technology

- No *clear choice* of a database system for data-based applications.

The above items will be addressed in the research and design of an artefact, which is the process model used for effective database development and design. The items listed will be combined to provide a complete DSS framework that defines the process, procedures, methods and techniques, and the selection of technology.

The philosophy and methodology for addressing the above challenges are explored in the following section.

2.6 Research Design

2.6.1 Approach

The concept of emergence was introduced after translating real-world challenges into an academic environment. This emergence is an inductive methodology that originated from repeated interaction between the real world (practice) and the body of knowledge (theory) [32].

New knowledge emerged by performing empirical experimentation (informed by theories and methods contained in the body of knowledge) in a controlled environment to characterise database performance behaviour when manipulating design parameters. Reflecting on this counterintuitive knowledge led to a pragmatic, mostly technology-agnostic approach to database design.

Therefore, the primary research approach is inductive, supported by DSR, focusing on creating a new methodology in the form of a decision support system.

Importantly, although following an inductive approach to produce design artefacts, it was important to follow deductive reasoning [33]. Deductive reasoning was used to test design

parameter impact during empirical experimentation, while inductive reasoning led to the decision support system.

Also, qualitative and quantitative approaches were used during this research, resulting in a mixed-methods approach [34]. A qualitative approach considered existing database modelling techniques and documentation. In contrast, a quantitative approach was followed while performing empirical experimentation and analysis.

2.6.2 Strategy

Since a large portion of experimentation was dependent on a research and development (R&D) process, all methods relating to R&D were applied. These methods fit well within the DSR paradigm (not a research method). The research methods followed in this study are as follows:

***Documentary Research*:**

Different research topics relating to the problem were identified and researched [33]:

- Databases:
  - Relational Databases;
  - NoSQL Databases;
  - Database Design;
  - Data Modelling;
  - Data Transformation.
- Application Development:
  - Cloud Processing;
  - Life cycle of an Application;
  - Time to Market;
  - Cost of Development;
  - Training and Education.

The systems engineering process was specifically researched as it informed the research process based on a full life cycle approach [23].

***Experimental Research*:**

Experiments were conducted using defined and controlled computer benchmarks [33]. Specific characteristics of the real-world operational database capabilities were obtained and used to define the parameters used in the empirical experiments. Results were used to evaluate system performance and characterise typical database behaviour to model design and operations changes. These results were then used to inform the new design methodology (decision support system).

To overcome the research challenges, the following steps were identified in the literature and completed by this research project, also grouped according to the framework items:

(1) Process

- Use a full life cycle systems approach to data-based application design;
- Design a decision support framework for database design.

(2) People & Procedures

- Use a full life cycle systems approach to data-based application design.

(3) Methods & Techniques

- Evaluate different conceptual modelling techniques;
- Use best practices to model equivalently in both types;
- Critically evaluate and propose criteria.

(4) Tools & Technology

- Determine strengths and weaknesses of each database.

2.6.3 Systems Engineering

Systems engineering (SE), as a methodology, focuses on form, fit, and function. It defines tools and methods for a function's specification, design, creation, operation, and maintenance. SE formed an essential part of the decision support system's pragmatic approach. It consists of the following relevant elements [35]:

- Identification and quantification of goals and objectives of a system;
- Functional and performance requirements establishment;
- Concept design alternatives compliant with the requirements;
- Performance, cost-benefit, trade-off analysis;
- Selection, implementation and deployment;
- Design verification;
- Post-deployment assessment of goals.

The systems engineering process is divided into the following two main life cycle phases and activities [23]:

- System acquisition:

  - System design;
  - Implementation;
  - Test and integration;
  - Construction or production;
- System utilization:
  - Operation;
  - Maintenance and support;
  - Upgrade;
  - Retirement.

Proper database modelling and design positively impact all the phases following it. Especially if future improvements, expansion and growth have to be considered, a more maintainable, scalable and agile design has to be made.

A term often encountered is *systems thinking* which can be defined as conceptualising real-world phenomena with models. Models are characterised according to assumptions made about [36]:

- Purpose;
- Cause-and-effect;
- Certainty and determinism;
- Inclusion of human and social concerns;
- Presence of feedback, adaptation and learning.

It is precisely this form of systems thinking which is foundational to this research process. UML tools can be used to great effect alongside systems engineering and are, therefore, good modelling, design and documentation aides for this thesis [37].

2.7 Research Evidence

Evidence was obtained in different forms. In the DSR context, evidence was obtained from empirical data model experiments, which altered parameters in the preliminary design phase and measured the impact in a controlled environment.

The emergence of knowledge from these experiments (theoretical space) leads to the creation of a new artefact (decision support system). This artefact was applied in a real-world use-case experiment, which was validated to produce favourable improvements as predicted.

Furthermore, the completed RVM shows the traceability for verification and validation from the original information sources to the final solutions.

2.8 Research Scope

A well-defined scope is necessary for productive research.

Due to the vast amount of database technologies and products, only the best-of-breed open-source database technologies had to be chosen for this research.

Since relational databases and document databases are the two most popular and mature DBMS categories, only MongoDB and PostgreSQL were to be considered [38]. MySQL does not provide a tabular (relational) ability to store columnar data[39] and was therefore excluded from the scope.

Only one standard, trusted, representational and complex real-world test case had to be used for validating the DSS.

2.9 Research Conclusion and Contributions

Contributions that DSR generates are listed as follows [40]:

- Identification of a relevant problem;
- Demonstration that an adequate solution does not exist in the public domain;
- Synthesis of a novel artefact;
- Rigorous evaluation of the artefact;
- Communication of the added value from the artefact;
- Dissemination of the research output.

One type of design artefact is a *process* or *method* artefact, the type of artefact this research generated. Evaluating new technologies is also an essential aspect of DSR, and this thesis evaluated NoSQL, traditional relational, and hybrid data modelling [27].

This research project makes the following contributions:

- Process of translating UML class diagrams to a document-oriented data model;
- Characterisation of empirical database performance and design parameters;
- A pragmatic strategy for hybridisation of relational and non-relational data models;

- A design methodology for hybrid data-based applications;
- Critical evaluation of modern and traditional database technology;
- Systems-engineering perspective to data-based application design;
- A decision support system for database modelling.

2.10 Summary

This chapter has thoroughly elucidated the research design for this thesis by stating the problem and defining the philosophy, paradigm, and methodology followed. Furthermore, the roadmap and framework used by this research project were illustrated and discussed.

To summarize the research challenges using the QRM framework, the challenges are shown in the table below, linked to the sources that validate the challenges as indicated in Table 2.2. This table will be used to direct the research process:

Table 2.2: Concept research challenges and sources of validation

Concept Research Challenges / Information Sources	No *clear choice* of a database system for data-based applications	*Design and implementation* criteria for each model type are unknown	*Conceptual modelling* approaches are almost exclusively for relational models	No equivalent *logical modelling* approach for both data model types	Application and data model design are usually *separated*	Full *life cycle* approach not considered in methodologies	There is a need for a definitive guide or *strategy* for *designing and planning* data-based applications in the ICT full life cycle
Database documentation	×	×		×		×	×
Database design literature	×	×	×	×	×		×
Application design literature					×		×
Systems Engineering literature					×	×	×
Related work and rationale	×	×	×	×			×
Personal observations	×	×	×	×	×	×	×

Although this chapter summarises various aspects described in depth in later chapters, it is essential to show how the research fits together in the light of the philosophy, paradigm and methodology.

2.11 Conclusion

This chapter defined the research management process, paradigm, and specific approach to address research challenges in a systematic way. Different world views were identified and discussed to provide context to the research problem – these are critical to understand as the correct ontology is required for a generalised solution to be produced. The research is encapsulated in a Systems Engineering environment, which is the most appropriate framework for multi-disciplinary system development (adapted to information technology systems), in this research.

The following system elements were considered as part of a DSS solution context, namely (i) process, (ii) people and procedures, (iii) methods and techniques, and (iv) tools and technology. The DSS process will define clear high-level data-based application development tasks, design procedures will be derived from processes and implemented by database developers and designers, database-specific methods and techniques will be applied during the procedure, and database technology's characteristics must be known in order to meet application requirements.

The research question: *"How does a real-world data-based application benefit from pragmatic design and implementation?"* had to be answered. This was achieved by performing a focused literature study, conducting DBMS characterisation experiments, developing a DSS process model, and validating the DSS model against a test case.

If you steal from one author it's plagiarism; if you steal from many it's research.

Wilson Mizner

Chapter 3

Literature Study

Introduction

As part of the QRM process, it was necessary to identify literature focus areas that confirm the research challenges and assist with constructing a solution. To this end, a Research Validation Matrix (RVM) links research challenges to literature focus areas and concept research solutions.

Table 3.1 shows the RVM with allocations from literature topics to research challenges. It also shows how research focus areas/topics relate to the research solutions in this chapter. Note that the arrows in the matrix are used to show a link between the research topic and research challenge (pointing up) and the research topic and research solution (pointing down).

3.1 Databases

Databases store and share a collection of related end-user data and metadata to meet information needs. The former is raw, unformatted data and the latter is data describing the end-user data. Metadata may, for instance, describe the relationships linking various pieces of data. Database management systems (DBMSs) were used for many applications as early as the 1960s. DBMSs are systems implementing and managing a database and serving as an intermediary between the database and the user [3, 41].

Many variations of DBMSs exist, but for this thesis, they can be grouped into two main categories: (i) Relational (SQL) and (ii) NoSQL [42].

Big data, a relatively recent term, is used to describe vast amounts of data originating from the increased Internet accessibility of the modern world. This data can be structured, semi-structured or even unstructured. Processing big data requires speed, flexibility, and,

Table 3.1: Literature sources that support concept research challenges and solutions

	Concept Research Challenges / Literature Topics	No *clear choice* of a database system for data-based applications	*Design and implementation* criteria for each model type are unknown	*Conceptual modelling* approaches are almost exclusively for relational models	No equivalent *logical modelling* approach for both data model types	Application and data model design are usually *separated*	Full *life cycle* approach not considered in methodologies	There is a need for a definitive guide or *strategy* for *designing and planning* data-based applications in the ICT full life cycle
Databases	Relational Databases	↓↑	↑		↑			↑
	NoSQL Databases	↓↑	↑		↑		↓↑	↑
	Database Design	↓↑	↓↑	↓↑	↓↑	↓↑	↑	↑
	Data Modelling		↑	↓↑	↓↑	↓↑	↓↑	↓↑
	Data Transformation							↓↑
Application Development	Cloud Processing							↓↑
	Life cycle of an Application						↓↑	
	Time to Market	↓↑				↓		↓
	Cost of Development	↓↑				↓		↓
	Training and Education	↓↑				↓		↓
	Systems & Systems Engineering	↓↑				↓		↓

	Literature Topics / Concept Research Solutions	
	Determine strengths and weaknesses of each database	↓↑
	Critically evaluate each model type and propose criteria	↑
	Evaluate different conceptual modelling techniques	↓↑
	Use best practices to model equivalently in both types	↑
	Use a full life cycle systems approach to data-based application design	↓
	Design a decision support framework for database design	↓

most importantly, scalability. Scalability is often reached by distributing data [43].

Big data has become intertwined with cloud computing. Selected sources express that traditional relational databases are not well suited for the cloud or big data. NoSQL has emerged to address the issues associated with big data [12, 44], namely: availability, scalability, and flexibility.

Other sources claim that relational databases also apply and can scale well, as seen with big-data giants *Facebook*, *MySpace* and *Twitter*. Both relational and NoSQL databases must be applied to the correct context to be scalable [12, 39, 45, 46].

MongoDB will be used for the NoSQL database due to the support for auto-sharding, indexing, aggregation and a high-level query language [47]. It also supports multi-document transactions, which is required to compare to the relational paradigm [48, 49].

For relational databases, this thesis will focus on the popular SQL-based ORDBMS (Object RDBMS), PostgreSQL. Recent developments exist to create an auto-sharding and horizontally scalable PostgreSQL cluster while maintaining ACID and transactional capability [50, 51] and will be used in this thesis.

The rationale is that a comparison of the relational paradigm to a modern one will be made for scalable cloud applications and therefore requires a database that supports a modern approach.

ORDBMSs can also model data with a modern paradigm comparable to NoSQL. Furthermore, PostgreSQL provides the `jsonb` column type, which is an efficient yet flexible NoSQL interface available in concert with its traditional roots [51]. For this reason, PostgreSQL is chosen to represent the relational databases.

3.1.1 Relational Databases

Relational databases use tables to represent data, and attributes of different data types are stored per column of a specific row. Relational databases use meta-data to describe relationships and constraints between tables (in essence, the rows of the tables) [2, 52].

Most legacy RDBMSs provide a query language known as SQL (Structured Query Language), which is used to manipulate and query the data [53].

RDBMSs can perform transactions due to the ACID attributes they provide. Transactions are an all-or-nothing database action, performing multiple queries and actions. The transactional ability is one of the most compelling features of modern RDBMSs [45, 54].

A caveat of traditional relational database design is that one must use normalization during the design phase to ensure consistency and integrity in a model. This normalization brings about a requirement to pre-design the schema in a semi-fixed way. Minor changes

are possible, but it is not flexible or performant. The pre-designed schema has to contain all the fields that would *ever* be required (as far as possible), resulting in empty fields and inefficient storage, leading to poor performance [55].

Different forms of normalization exist, *first* through to *fourth*, each moving further away from data redundancy, producing the requirement for more table joins. Heavily normalized data is intricately coupled with relationships to JOIN tables, and retrieving a full view of the data requires joining multiple tables [41].

Normalization forms a central and core part of the traditional relational database modelling paradigm. If normalization is not used, integrity problems can occur during updates, insertion, and deletion [2, 41].

Integrity and consistency could be addressed at the application level in a denormalized state [1].

Considering the cloud, it is demonstrable that scalability decreases as joins increase, as the strong coupling between tables makes it difficult to partition and slow to execute [12, 53, 56].

Traditionally, RDBMSs scale vertically by adding more storage, memory, and processing power to a single computer. This form of scaling, in contrast to horizontal scaling, has an upper physical limit and potential cost implications. Horizontal scaling is more flexible than its counterpart, albeit more complex [57–59].

RDBMSs can scale horizontally by relaxing ACID properties, making it challenging to allow transactions and joins on distributed data at the database level. However, this can also be addressed at the application level [16].

Tumblr has released a presentation on horizontal scaling for MySQL, demonstrating how to shard the database, thereby increasing write scalability [60].

Storage engines such as the Spider storage engine have also been released, capable of sharding, high availability and transactional capability (via two-phase-commit) [61].

3.1.2 NoSQL Databases

Four general types of NoSQL databases exist, namely key-value, column-oriented, document and graph [62, 63]

NoSQL systems usually have a shared-nothing architecture, which lends itself naturally to horizontal scalability [64].

Key-value stores make use of a key mapping to opaque values. These databases are more suited to simple operations based on key attributes and are often used as a caching layer for slow SQL queries. Examples of key-value stores are *Voldemort*, *Redis* and *Membase*

[65].

Cassandra, *BigTable* and *Hbase* are examples of column-oriented databases. Groups of related data are stored in rows of columns and column families. Each row can be flexible in terms of the columns used, and columns can also be supersets of other columns, as seen in Figure 3.1 [66].

Row Key	Column Families			
CustomerID	CustomerInfo		AddressInfo	
1	CustomerInfo:Title	Mr	AddressInfo:StreetAddress	9999 500th Ave
	CustomerInfo:FirstName	Mark	AddressInfo:City	Bellevue
	CustomerInfo:LastName	Hanson	AddressInfo:State	WA
			AddressInfo:ZipCode	12345
2	CustomerInfo:Title	Ms	AddressInfo:StreetAddress	888 W. Front St
	CustomerInfo:FirstName	Lisa	AddressInfo:City	Boise
	CustomerInfo:LastName	Andrews	AddressInfo:State	ID
			AddressInfo:ZipCode	54321
3	CustomerInfo:Title	Mr	AddressInfo:StreetAddress	999 500th Ave
	CustomerInfo:FirstName	Walter	AddressInfo:City	Belleveue
	CustomerInfo:LastName	Harp	AddressInfo:State	WA
			AddressInfo:ZipCode	12345

Figure 3.1: Column-oriented data

Graph databases such as *Neo4j* and *AllegroGraph* store data in nodes of key-value pairs. Nodes are attached with relationships, thus building a graph model [63]. These databases are well suited for handling heavily interconnected relationships, such as social networks, pattern recognition, dependency analysis, recommendation systems and pathfinding problems [67]. Figure 3.2 shows an example of such a model [68, Fig. 3].

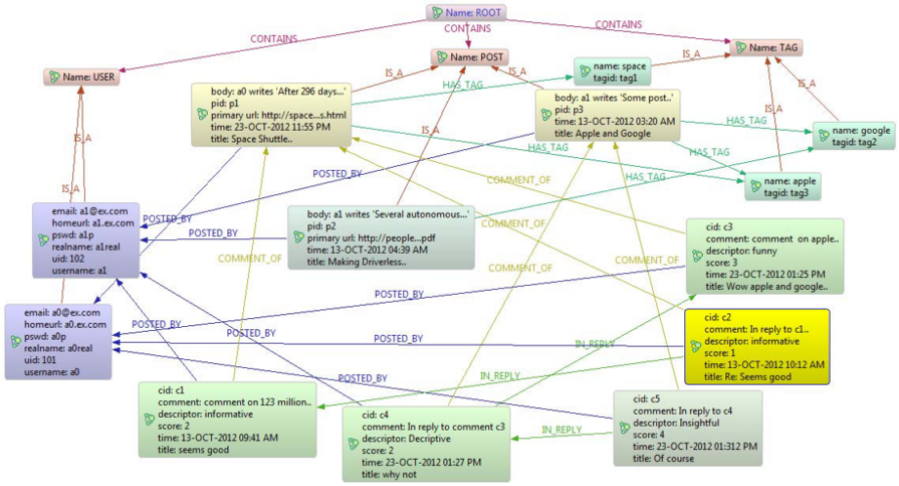


Figure 3.2: Graph model

Document databases are also key-value-based but incorporate complex structures as well. Document formats such as XML and JSON closely reflect the data model used by these databases [69]. MongoDB (taken from *humongous*) is a document-oriented database that

started in 2007 [70] and will be the focus of the NoSQL study.

According to MongoDB, it can be used in most use cases that relational databases can be used [47], except it is not ideally suited for complex multi-row transactions or applications built around relational models or SQL [71].

Sharding data to multiple nodes can massively scale MongoDB clusters. For this to work, the data must be sharded on a shard-key [72]. It is also essential for performance that joins are mitigated [3, 73]. In order to achieve this, MongoDB allows embedding data, in effect pre-joining the database [74].

Embedding is not without caveats. It brings about a consistency and integrity problem when the embedded data has to change, causing potentially significant fan-out updates on relevant documents, leading to eventual consistency [75].

MongoDB can also store relational data by embedding a reference (foreign key) into the relating document. However, embedding in MongoDB does not join at the database level in the traditional sense [10, 76, 77]. Recent additions to MongoDB allow left outer joins to be performed with the aggregation pipeline and the `$lookup` operation [78].

Operations on a single document are atomic, which provides the ability to implement two-phase-commit (2PC) transactions, albeit at a performance cost [11, 79].

Depending on the implementation of the data model, strong consistency can be achieved with MongoDB [80].

There is also full ACID capability with multi-document, inter-shard transactional support since version 4.2 [48, 49].

3.1.3 Database Design

In designing a database for distributed systems, it is vital to ensure how the data will be represented. The representation and access patterns will be subject to the CAP theorem, described below, and will impact the design, especially with eventual consistency [81].

As part of data model design, normalization minimizes data duplication, thereby increasing consistency. This step is part and parcel of traditional relational modelling [82].

When designing a data model, it is important to represent the data and relationships in a structured fashion. Entity-Relationship Diagrams (ERD) and Unified Modelling Language (UML) class diagrams are the two most popular visual tools, and this thesis will focus on UML due to its overlap with software development [83].

Chen's Methodology

Chen has developed a foundational modelling framework for representing entities and relationships [84].

The Entity-Relationship (ER) model has become the *de-facto* standard data modelling framework. The ER methodology is summarised as follows [3, 20, 84, 85]:

1. Conceptual modelling;
    1.1 Requirement gathering;
    1.2 Specification;
    1.3 Modelling: ER diagrams or UML class models;
2. Logical modelling;
    2.1 Normalization processes;
    2.2 Table structures;
    2.3 Primary keys;
    2.4 Database schema is the output (not DBMS specific);
3. Physical modelling.
    3.1 Specific database structures (DBMS specific);
    3.2 Secondary indexing;
    3.3 Consistency;
    3.4 Constraints;
    3.5 Physical storage design (disk management and organization);
    3.6 Access control and rules;

Often the modelling process happens on the entities and relationships alone and does not consider the underlying system or transactions required. Furthermore, the output of this process is then the artefact used to specify the requirements for applications that would use the DBMS [12, 20, 85].

CAP Theorem

In 2000 Eric Brewer made a conjecture that a web service cannot guarantee all three of the following traits [86] namely, consistency, availability, and partition-tolerance.

It is empirically proven to be correct for asynchronous models. However, it is possible to balance consistency and availability using the asynchronous model [87].

Brewer clarified the theorem in 2012 by stating that partition tolerance cannot be neglected. He claims that NoSQL movements (BASE) prioritize availability above consistency, while ACID databases adhere to the contrary [88].

Furthermore, Brewer also stated that the CAP-theorem might be applicable on a microscale rather than on the whole system [88, 89].

A variation of CAP, called PACELC, is used to address some shortcomings for situations when partitions do exist and during normal operations. It works as follows [90]:

- When a ***P****artition* exists;
- Choose between ***A****vailability* and ***C****onsistency*;
- ***E****lse*;
- Choose between ***L****atency* and ***C****onsistency.*

According to this, MongoDB would be a PA/EC, trading off consistency for availability during partitions while under normal operations guaranteeing strong consistency rather than latency. ACID databases with traditional modelling would be PC/EC, or in other words, always preferring consistency [90].

Furthermore, Stonebraker also claims that CAP isn't accurate during the following events, and the rarity must be taken into consideration in the application [91]:

- Application errors (ex. incorrect updates);
- Repeatable DBMS errors (Bohr bugs);
- Disasters (physically destroying a cluster).

According to Gilbert and Lynch, an application must segment the consistency and availability trade-off by examining the data requirements, operations, functions, users, and hierarchies [92].

Bailis et al. have demonstrated that it is possible to reach HAT (highly available transactions). They developed a proof-of-concept system to reach high availability and consistency in partition-prone environments [93].

Some systems use data that is CALM or *consistency as logical monotonicity.* This concept states that programs growing without retracting data (monotonous data) can be developed using eventual consistency. It is then essential to test the data for monotony [94].

CRDTs, another data acronym, represents *convergent or commutative replicated data types.* These types of distributed data guarantee eventual consistency even with asynchronous replication [12, 94].

Entity Attribute Value

The Entity Attribute Value schema (EAV) was designed for flexibility in relational databases. It uses a table containing two foreign keys (one for the attribute and one for the value) and then joins at least three tables [11].

EAV is a design pattern, originally for relational databases, using self-joins to gain schema flexibility but exhibits poor performance. A rule of thumb is to have only a dozen or fewer tables per query [54].

Various sources claim that EAV is a universal design pattern not suited for performance or scalability but is useful when a very high quantity of parameters are used to describe an object, but very few apply to one [95, 96].

Normalization and Denormalization

Normalization groups attributes into refined structures, yielding stable structures diminished in update anomalies and maximal data accessibility [97]. Normalization does not improve performance, ease of use, or scalability [98, 99].

The three fundamental normal forms can be obtained through the following steps/rules [3]:

First Normal Form (1NF):
Remove repeating groups of attributes and create rows without repeating groups

Second Normal Form (2NF):
Remove partial key dependencies

Third Normal Form (3NF):
Remove transitive dependencies.

Denormalization is therefore defined as the reduction of the degree of normalization in order to improve query performance [100, 101].

The process of denormalization, in table-based designs, can use one or more of the following patterns [97]

Collapsing tables In essence, pre-joining tables.

Splitting tables Using partitioning instead of multiple table relations.

Using redundant columns Adding columns of frequently required related columns.

Derived attributes Attributes are pre-derived and stored in a column.

Using views Views can provide a pre-joined table.

A novel case of denormalization was demonstrated by Lee and Zheng. They automated the process by reducing all relationships in a SQL database into a single NoSQL-like table, increasing the performance by up to 48% in their use cases [102].

Furthermore, Kanade and Gopal proposed a hybrid data model to show that a best-of-both approach between normalized and denormalized can obtain better results than purely normalized in both MySQL and MongoDB [18].

PostgreSQL, being object-relational, supports the ability to denormalize by using user-defined column and array types [51].

Map/Reduce

Map/reduce is a big-data paradigm used to harvest and process data. It consists of, at least, a *Map* function and a *Reduce* function. The paradigm splits data into smaller chunks (Map), distributes the chunks to worker nodes for processing, and then combines and collects the results (Reduce) [103].

The Map function receives a key and value (a tuple or a document) and emits a key and list of homogeneous values. The Reduce function receives a key and a list of values and emits a list of homogeneous values. This is shown below [104]:

$$\text{map}(k1, v1) \longrightarrow \text{list}(k2, v2)$$
$$\text{reduce}(k2,\text{list}(v2)) \longrightarrow \text{list}(v2)$$

The advantage of Map/Reduce is that it lends itself to distributed processing in the cloud. It is highly scalable and has been used in simple and complex use cases [105–107].

MongoDB supports Map/Reduce functionality which has the following benefits [11, 48] namely, flexibility, scalability, efficiency, and fault tolerance.

However, it does not inherently support high-level query languages such as SQL and applications have to be developed for each task [108, 109].

With the introduction of MongoDB's aggregation pipeline, MapReduce has become somewhat redundant, as aggregation pipelines support any MapReduce functionality with higher performance and easier interface [48].

Two-phase Commit

Multiple document transactions can be implemented by a method called two-phase commit or 2PC. This type of transaction makes use of optimistic concurrency methods to obtain

consistency. To implement this in MongoDB is quite complex, but the same consistency can be reached as a full ACID database[1] [110].

The same type of transactional capabilities can be added to any application by implementing a form of 2PC and is not exclusively for databases [111].

3.1.4 Data Modelling

When considering the differences in data representation in RDBMSs and NoSQL, it is essential to model data accordingly [112].

UML (Unified Modelling Language)

UML is an industry-standard notation suite used for object-oriented software systems. It contains methods and tools used during the whole conceptual design process. Among other techniques, it provides the following [113]:

- Use-case diagrams: shows actors and actions;
- Activity diagrams: a form of a flowchart;
- Sequence diagrams: shows ordered sequence of events and interactions;
- Class diagrams: relationships, attributes and methods are shown.

UML is often used to conceptually model database relationships and activities [114, 115].

The phases of an application's life cycle up to the implementation phase can be documented within UML, and the remaining phases benefit from the process [116].

Use-case diagrams offer insight into the requirement specification needed for designing and implementing all elements of a software system [117].

Activity diagrams describe hierarchies of procedures in the form of a control and flow model. It is related to the sequence diagram, which models the interaction and message exchanges between elements [118].

Static structures of classes in a system are expressed with class diagrams. They represent the internal structure as well as relationships between classes. This diagram lends itself perfectly to the database modelling and application design world [119, 120].

Combining the static and dynamic elements of UML, one can derive a model that helps with problem-solving by giving a complete visualization of the elements at different levels of abstraction [116, 121].

[1] This is in fact higher than the transactional promises of an RDBMS without explicit locking

Embedding and Referencing

Relationships can be modelled as referenced (the traditional paradigm) or embedded data (the modern paradigm). MongoDB and PostgreSQL can model data in these two ways [122]. Combinations of these two can be used as well.

Referencing, in this context, means having a copy of the *unique identifier* of the related entity to look up the dependency uniquely and singularly. These are called foreign key references in relational databases. It is possible to store an array of references in cases where appropriate (for low cardinalities) [6, 11].

Pure referencing always requires a lookup or joining with another table or collection to populate the attributes of the entity referred to when performing queries. It is this fact that impacts read performance [18].

Embedding has many options in the context of the existence of an entity [74]:

A An entity might exist as a top-level strong entity and be referenced elsewhere (no embedding);

B A weak entity might exist purely due to the existence of another entity (pure embedding);

C An entity might exist as a top-level strong entity and be embedded (fully or partially) within other entities (weak relationships; hybrid embedding);

In the case of **A**, there is no embedding, and the data is fully normalized. In the case of **B** the data is fully denormalized (exists purely within other entities). Finally, in the case of **C**, the entity exists as a standalone, or strong, entity but is duplicated in part or whole within related entities.

It is important to note that case **A** does not inhibit the capability to model relationships with references (foreign keys), and a combination of embedding and referencing is in essence a hybrid modelling approach.

1-1 (One-to-one)

When only one instance of an element is related to one other instance, it is considered a one-to-one relationship, as shown in Figure 3.3. It is rare in relational databases but is often employed to enforce constraints, such as when there can only be one head employee of a department even though a department has a one-to-many relationship with employees [41].

One-to-one relationships can be modelled in MongoDB by embedding or referencing the data either in the employee document, the department document or both, as shown in

Figure 3.3: One-to-one relationship

Listing 3.1 [6].

```
// employee collection
{
  "_id": ObjectId("5b273446aa567a2ee930235"),
  "name": "John Doe",
  // Head of this department
  "heads": ObjectId("6b273446aa567a2ee930346"),
}
// department collection
{
  "_id": ObjectId("6b273446aa567a2ee930346"),
  "name": "Engineering",
  // Headed by this employee
  "headed_by": ObjectId("5b273446aa567a2ee930235"),
}
```

Listing 3.1: MongoDB One-to-one relationship

It should be noted that eventual consistency could be employed by embedding some attributes of the dependency on either or both ends, reducing the need for joins and multiple database round trips. This form of embedding is especially useful for rarely changing data by not requiring as much fan-out on changes.

Similarly, PostgreSQL can model the relationship equivalently as shown in Listing 3.2 with a bidirectional one-to-one relationship. Note that bidirectional references are not required and could be reduced to a single reference in any of the two tables, depending on the query required.

1-M (One-to-many)

A typical example of a 1-M relationship is an employee working for a company. Typically, an employee only works for one company, but a company can have many employees, as shown in Figure 3.4 [123].

In a traditional normalized relational database, this relationship should be modelled as two tables, employee and company, where the employee will have a foreign key referring to the company that employs him/her [41].

```
CREATE TABLE employee (
    _id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name VARCHAR
);

CREATE TABLE department (
    _id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name VARCHAR,
    headed_by uuid REFERENCES employee
);

/*Required to be added after the fact
because department didn't exist yet*/
ALTER TABLE employee ADD COLUMN heads uuid REFERENCES department;

/*Automatically assign John Doe as Engineering head*/
WITH emp_rows AS (INSERT INTO employee (name) VALUES
            ('John Doe') RETURNING _id)
INSERT into department (name, headed_by)
SELECT 'Engineering',_id from emp_rows;

/*Update employee heads for bidirectional referencing*/
UPDATE employee SET heads =
    (SELECT _id from department
        WHERE headed_by = employee._id);
```

Listing 3.2: PostgreSQL One-to-one relationship

Figure 3.4: One-to-many relationship

With a modern paradigm, such as portrayed by MongoDB, there are a few options [6]:

1. Fully normalized with reference;
2. Fully embedded in an employee;
3. Fully embedded in a company;
4. Partially embedded[2] in an employee;
5. Partially embedded in a company.

Generally, embedding is better for fast reading of small or slow-growing documents that do not change regularly or where eventual consistency is acceptable *and* the data fields are often required during a query. In contrast, references are better for fast writing of large documents or volatile data, where consistency is paramount, or documents grow large, or data fields are often excluded in queries or change regularly [6].

In this example, since employees are unbounded, the best scenario would be to normalize fully. The employee documents may, however, embed information about the company if required since an employee only works for one company. This would require a change in the company's data and the application to accept stale or eventually consistent data (when updating the embedded data without a transaction). Depending on the data volatility and query patterns, it may be a good trade-off between writing and reading speed.

This thesis will use the definition of one-to-many for a relationship that is unbounded on the many side.

For a semi-embedded model in MongoDB, the results could look like Listing 3.3. PostgreSQL could model the same with the DDL (Data Definition Language) shown in Listing 3.4. Note here that by using the user-defined type (`company_reference`), foreign key constraints cannot be checked by the database. It could also be modelled directly as a reference/foreign key field as in traditional relational models.

1-m (One-to-few)

A particular case of one-to-many relationships is what this thesis will call one-to-few [6]. This type of relationship is bounded to a certain degree. An example would be a user having multiple email addresses. In normalized relational models this should be modelled by two tables, one for the user and one for emails, however, a user will typically only have a few additional email addresses. In NoSQL and modern paradigms, this could be modelled by an embedded array of email addresses [76].

[2]Partially here refers to embedding *some* fields often queried or joined

```
// company collection
{
  "_id": ObjectId("6b273446aa567a2ee930346"),
  "name": "ABC Engineering"
}
// employee collection
{
  "_id": ObjectId("5b273446aa567a2ee930235"),
  "name": "John Doe",
  // Head of this department
  "company": {
      ref: ObjectId("6b273446aa567a2ee930346"),
      name: "ABC Engineering",
  }
}
```

Listing 3.3: MongoDB One-to-many relationship

This distinction between many and few is critical since it could impact relational databases' performance due to an additional join operation to retrieve a user's entire record. In the case of the user example, strong consistency is not lost either.

MongoDB could model emails simply as an array of strings, and PostgreSQL could do the same as shown in the Listing 3.5. These are simple examples, but complex data can easily be embedded in either database.

M-N (Many-to-many)

Many-to-many relationships can be demonstrated by the example of players in a sports team. Many players play in one team, and one team consists of many players. This relationship cannot be modelled physically in a normalized relational database without adding a join table or contract, for example, as shown in Figure 3.5 and Figure 3.6. [2]

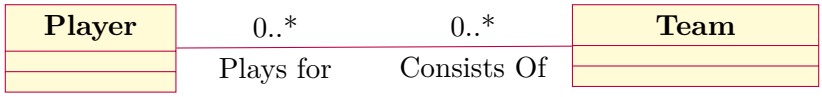


Figure 3.5: Conceptual M-N relationship

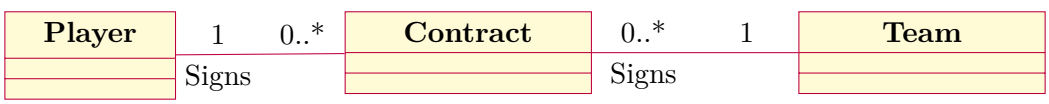


Figure 3.6: Join table for M-N relationship

```
CREATE TABLE company (
    _id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name VARCHAR
);

/*Stub to embed name and reference*/
CREATE TYPE company_reference AS (
    ref uuid,
    name VARCHAR
);

CREATE TABLE employee (
    _id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name VARCHAR,
    worksfor company_reference
);

/*Automatically assign ABC Engineering
as John Doe's company*/
WITH company_rows AS (INSERT INTO company (name) VALUES
            ('ABC Engineering') RETURNING _id,name)
INSERT into employee (name, worksfor)
SELECT 'John Doe',(_id,name)::company_reference from company_rows;
```

Listing 3.4: PostgreSQL One-to-many relationship

```
CREATE TABLE users (
    _id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name text,
    email text[]
);

INSERT INTO users (name, email) VALUES
        ('John Doe', '{"joe@mail.com","joey@example.com"}');
```

Listing 3.5: PostgreSQL One-to-few embedded relationship

MongoDB can follow the same approach by employing a join collection, or one could embed every contract in a player directly. This would optimize the query patterns by eliminating an additional join. This thesis will refer to this relationship type as many-to-few (many players will play for a few teams or sign few contracts) [6].

Listings 3.6 and 3.7 show a many to many(few in this case) embedded representation in MongoDB and PostgreSQL respectively. An alternative representation for PostgreSQL using `jsonb` instead of user-defined type tables is shown in Listing 3.8, demonstrating its capability to implement a modern paradigm for data modelling.

```
// Team collection
{
  "_id": ObjectId("6b273446aa567a2ee930346"),
  "name": "Tigers"
}
// Player collection
{
  "_id": ObjectId("5b273446aa567a2ee930235"),
  "name": "John Doe",
  // "Join" collection existing purely as embedded data
  "contracts": [{
      ref: ObjectId("6b273446aa567a2ee930346"),
      name: "Tigers",
      started: "2019-01-01T23:28:56.782Z",
      ended: "2021-01-01T23:28:56.782Z",,
    }, {
      ref: ObjectId("6b273446aa567a2ee930346"),
      name: "Tigers",
      started: "2022-01-01T23:28:56.782Z",
      ended: null,
    }
  ]
}
```

Listing 3.6: MongoDB Many-to-many relationship

These examples are a few of many alternative models that would be possible, each with strengths and shortcomings.

```
CREATE TABLE team (
    _id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name text
);

CREATE TYPE contract AS (
    ref uuid,
    name text,
    started timestamp,
    ended timestamp
);

CREATE TABLE player (
    _id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name text,
    contracts contract[]
);

WITH team_rows AS (INSERT INTO team (name) VALUES
        ('Tigers') RETURNING _id, name)
INSERT INTO player (name, contracts)
SELECT 'John Doe',ARRAY[
  (_id, name,
   TIMESTAMP '2019-01-01 23:28:56.782Z',
   TIMESTAMP '2021-01-01T23:28:56.782Z')::contract,
  (_id, name,
   TIMESTAMP '2022-01-01 23:28:56.782Z', NULL)::contract
] from team_rows;
```

Listing 3.7: PostgreSQL Many-to-many relationship

Data Transformation

Data transformation refers to transforming data from one representation to another, for example, from relational to NoSQL. This is widely referred to as extract-transform-load or ETL. Various tools already exist to transform SQL-based data into MongoDB documents [124].

MongoDB can be accessed from within PostgreSQL by making use of the MongoDB foreign data wrapper plugin, `mongo_fdw`, provided by EnterpriseDB. This plugin allows collections to be read and written within the PostgreSQL context. The wrapper transparently converts queries to and from the MongoDB context while providing a foreign table-based

```
CREATE TABLE team (
    _id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name text
);

CREATE TABLE player (
    _id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
    name text,
    contracts jsonb
);

WITH team_row AS (INSERT INTO team (name) VALUES
        ('Tigers') RETURNING _id, name)
INSERT INTO player (name, contracts)
SELECT 'John Doe', ('[{
                    "ref": "' || _id || '",
                    "name": "' || name || '",
                    "started": "2019-01-01 23:28:56.782Z",
                    "ended": "2021-01-01T23:28:56.782Z"
                    },{
                    "ref": "' || _id || '",
                    "name": "' || name || '",
                    "started": "2022-01-01 23:28:56.782Z",
                    "ended": null
                    }]')::jsonb
from team_row;
```

Listing 3.8: PostgreSQL Many-to-many relationship with `jsonb`

interface in the PostgreSQL context [125].

3.2 Application Development

3.2.1 Roles and Responsibilities

According to Ponniah, the roles and responsibilities in IT can be grouped under the following [3]:

Business analysts Interact with database practitioners and serve as an intermediary between them and senior management.

Data modellers Create semantic models from requirements and specifications.

Database designers Develop views of data to be integrated into the logical design of a DBMS.

System analysts Derive process requirements and develop application specifications.

Programmer/analysts Develop applications with programming code that interact with the database.

Database administrators Control and manage (administer) the DBMS. Create the physical model from the logical model and apply any physical database definitions (indexing, schemas, keys). Implementation and organization of the physical requirements. Authorize access to the DBMS. Monitor usage. Perform maintenance. Consult other practitioners of the system. Define backup and recovery of the DBMS. Improve database performance.

From the above descriptions, it is clear that the traditional role of the programmer (application developer) is separated from the roles and responsibilities of the design, modelling and implementation of the database system — including database performance.

Some authors, such as Badia and Lemire and Meijer are concerned about this separation and the state of database development keeping up with modern times [12, 20].

3.2.2 Cloud Processing

Cloud processing is not new, as it has been used conceptually since the 1960s. Cloud brings about new-found security and safety for applications [126]. The term was coined in the early 1990s [127].

Cloud processing has several requirements for applications [128, 129]:

- Elasticity: scaling horizontally;
- Flexibility: adaptable data models and schemas;
- Fault tolerance: high availability.

To achieve this, some systems (NoSQL) have opted to provide a relaxed transactional capability. New ways are being invented to keep the transactional promises while building on scalable infrastructure [130].

Advantages of developing for the cloud include [127]:

- Reduced cost and overhead for preparation;

- Reduced overhead for new instances;
- Eliminating application wastage;
- Increased stability due to scaling out, falling over, upgrading, etc.

3.2.3 Life cycle of an Application

When developing an application, keeping the life cycle in mind is important. Typically an application has the following classical stages [131, 132]:

- Requirements Phase;
- Analysis Phase;
- Design Phase;
- Implementation Phase;
- Post-delivery Maintenance;
- Retirement.

The first three phases concerning data modelling have been addressed in paragraph 3.2.4. This section focuses on the post-delivery maintenance phase.

The post-delivery maintenance phase is a continual process that consists of corrective maintenance and enhancement (specification changes). This enhancement task is important to database modelling and design [131].

Schema evolution happens when a deployed application needs to make changes. Relational databases use complex migration scripts to update schemas, sometimes even dropping a database and re-creating it. The flexibility given by schema-less NoSQL databases makes schema evolution less troublesome [11, 133].

3.2.4 Time to Market

To quickly assimilate market needs, it is essential to have a short time to market. Tools and methods used during development should add agility and be chosen wisely [134].

Rapid prototyping can be used to generate artefacts rapidly, and the cloud environment allows for on-demand requisition of services [135].

3.2.5 Cost of Development

Development costs can be attributed to [131]:

- Salaries of development teams, managers, and support personnel;
- Hardware and software costs;
- Overhead costs: rent, utilities, and senior management salaries.

All of the above can be linked to the duration of a project, and thus development time will directly impact the project's cost [136, 137].

3.2.6 Training and Education

The complexity of a project, including the ability to learn new technology, can negatively impact a project's budget and duration [138].

Furthermore, the required experience level also directly influences the project [139].

NoSQL systems, for example, are generally easier to deploy and learn [140].

3.3 Case Studies

To better understand the detail of the study, a literature survey of possible use cases is discussed in this section. It is assumed that PostgreSQL is well suited for all use cases since it has been used for many years across all industries.

MongoDB claims to be applicable and suitable to all use cases except where complex multi-row transactions are required or SQL is specifically required. Typical use cases for MongoDB include [71]:

- Single view (denormalised view of related entities);
- Internet of Things (many connected devices often supplying time-series data);
- Mobile;
- Real-time Analytics (time-series data, with real-time aggregations);
- Personalisation (custom behaviour per individual);
- Catalogue (hierarchical data);
- Content Management (asset and meta-data driven).

According to Ganesh Chandra, content management systems are well-suited for MongoDB [141].

MongoDB provides use case studies and design examples for operational intelligence, product data management (e-commerce) and content management systems [78].

A study was done on MongoDB, among others, to demonstrate the viability of NoSQL databases in the IoT field [142].

3.3.1 Operational Intelligence

Operational intelligence is the technique used to generate actionable information from transactional data such as analytics and reporting [11].

Real-time analytics is a special type characterised by low latency (sub-second) and high availability (99.999%) [143].

3.3.2 Content Management System

Content management systems (CMS) provide storage and services for information assets and associated meta-data. Typical applications include blogs, websites, online publications and archives [144].

Some tasks associated with CMS are (i) meta-data and asset management [145] and (ii) storing comments on blog posts [146].

3.3.3 Internet of Things (IoT)

Internet of Things (IoT) is a concept used to describe a system where all physical assets and devices are connected and communicating [147].

Fields making use of IoT include [148]:

- Healthcare;
- Smart City: automation of utilities;
- Smart Home and Smart Metering: automation of homes;
- Video Surveillance;
- Automotive and Smart Mobility;
- Smart Energy and Smart Grid;
- Service Delivery;
- Networking and Communications Protocols (Machine-to-Machine or M2M);

- Big data;
- Sensor networks;
- User participation;
- Monitoring.

3.3.4 E-Commerce

Typical e-commerce concepts are product catalogue, inventory management and categories [78].

Catalogues typically contain lists of any item or asset, including meta-data about the item [149].

Inventory management is the handling of adding and removing items from shopping carts and eventually checking out or cancelling an order [150].

The hierarchical classification of items in a catalogue is part and parcel of e-commerce applications. Categories are used to portray this for each item, allowing for the indexing of products [151].

TPC-C is a standardized DBMS and cloud performance benchmark. The benchmark is based on complex online-transactional processing (OLTP) workloads. The business throughput is measured as the number of *new order* transactions processed per minute (TpmC) [152].

Advantages of the TPC-C test are summarised below [152–155]:

Realistic Workload TPC-C is designed to simulate a realistic OLTP (Online Transaction Processing) workload, which involves many short transactions that read and write small amounts of data. This type of workload is still common in modern database systems, and TPC-C provides a good representation of the types of transactions that these systems need to handle.

Standardization TPC-C provides a standardized and well-defined benchmark that allows for fair and accurate comparisons between different database systems. This is important for both vendors and customers, as it allows them to make informed decisions about which database system to use or purchase.

Continual Improvement TPC-C is a mature benchmark that has been in use for over twenty years. Over this time, it has been refined and updated to reflect changes in database technology and to ensure that it continues to provide an accurate representation of real-world workloads.

Although these workloads are typically run on relational databases, adaptations to benchmark MongoDB's performance have shown that recent MongoDB transactional support allows e-commerce workloads to perform and scale well on MongoDB [156].

3.4 Systems Engineering

Core aspects of SE had been discussed in Chapter 2, but some important elements relating to the SE life cycle as it compares to the ER methodology are critical to research and review.

A generic timeless life cycle is found in [23] that forms the underlying process of almost all development processes in multi-disciplinary environments. Although there are variations, the principle of first modelling and evaluating at the preliminary design level was found to be highly relevant and informative in the context of DBMS development. By using the underlying process as a validated reference to a mature process, the intention of full life cycle development was preserved as opposed to DBMS-specific development processes such as ER process [84] and the MongoDB-specific process found in [21, 22]. Following this important finding, the DSS was developed from first principles extensively based on experimental results, available literature, and creative input

Chen's methods for Entity-Relationship modelling overlap with the SE approach to design, but some important distinctions are evident. The ER methodology phases are summarised as follows [3, 20]:

1. Conceptual modelling;
    1.1 Requirements elicitation;
    1.2 Basic UML/ER diagrams;
2. Logical modelling;
    2.1 Rendering as tables;
    2.2 Reduction into normal forms;
3. Physical modelling;
    3.1 Describe in DBMS-specific language;
    3.2 Disk management, design and planning.

In contrast, SE has the following design steps and items, as depicted by the SE life cycle in Figure 3.7 [23]:

1. Conceptual design;
    1.1 Feasibility study;

1.2 Needs identification;

1.3 System planning;

1.4 Functional analysis;

1.5 Feasibility analysis;

1.6 System operational requirements;

1.7 Maintenance and support concept;

1.8 Performance measures;

1.9 System trade studies;

2. Preliminary design;

2.1 System functional analysis;

2.2 Synthesis and allocation of design criteria;

2.3 System and subsystem analysis;

2.4 System synthesis and definition (data prototyping, **modelling**, testing, prototyping);

3. Detail design;

3.1 System and product design;

3.2 System and product model development;

3.3 System prototype test and evaluation.

It becomes evident from the summary above that modelling in SE only occurs during preliminary design, which is in contrast with traditional database development, where UML or ER is used early during conceptual modelling. Furthermore, technology choices are also made at the end of the preliminary design phase and do not dictate the earlier phases.

There is an overlap between the SE preliminary design phase and the ER conceptual and logical modelling phases. This overlap provides grounds for comparison and interoperability between the two methodologies.

3.5 Summary

This chapter described two database paradigms, and the strong and weak points were discussed. It is evident from the literature that both DBMSs have a role to play and that it is critical for application developers to have sound strategies when designing applications that should scale and be agile.

MongoDB and PostgreSQL were the representative databases for the opposing paradigms. They both exhibit the required traits to perform a comparison, as shown in Table 3.2. Each

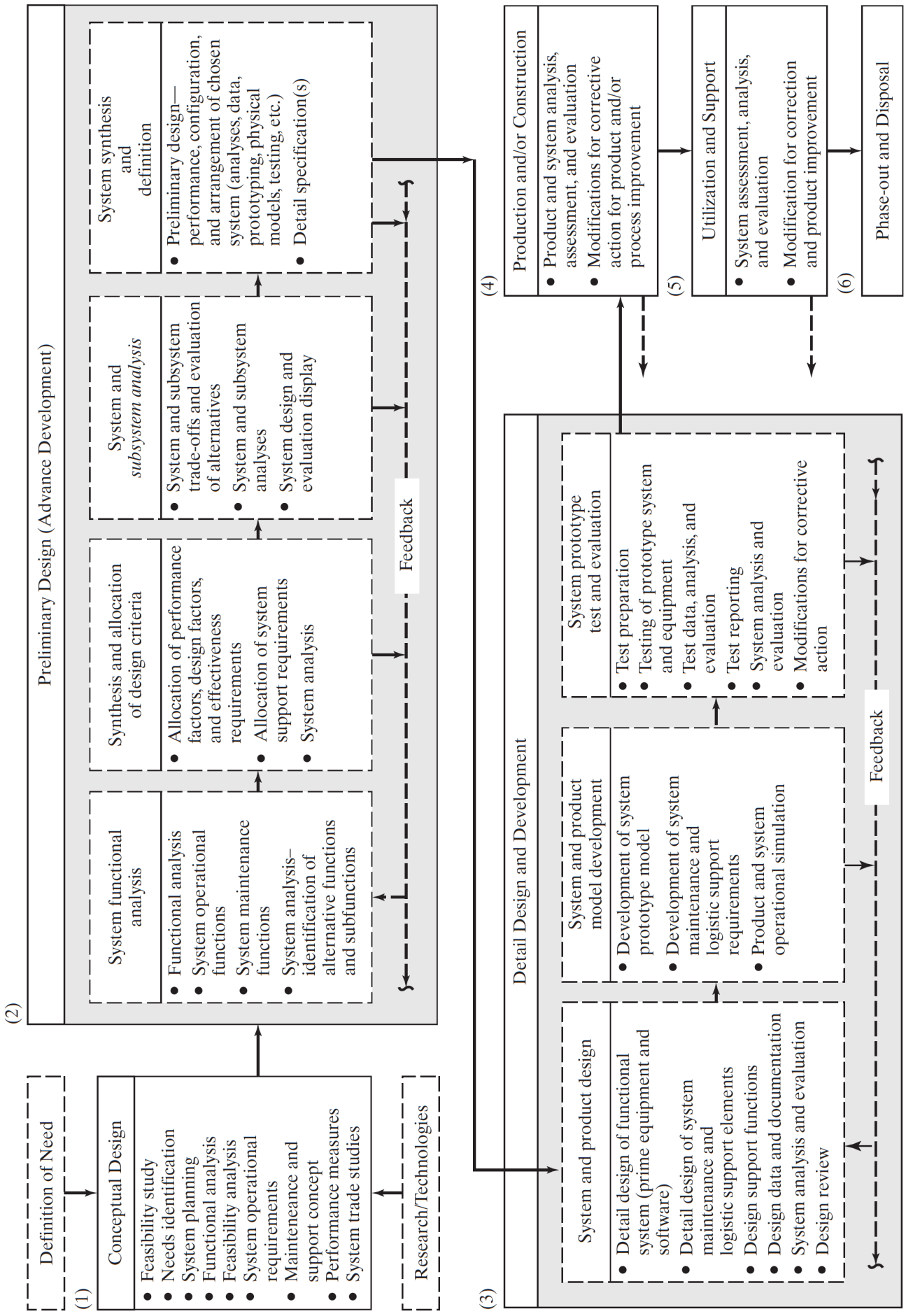


Figure 3.7: Systems Engineering life cycle processes [23, Fig. 3]

Table 3.2: A comparison of the equivalence of features in MongoDB and PostgreSQL

Comparison	PostgreSQL	MongoDB	Equivalent	References
Data modeling				
Entity Relational	Yes	Yes	Yes	[157, 158]
Entity Attribute Value	Yes	Yes	Yes	[48, 51]
Unstructured	Yes	Yes	Yes	[77, 158]
NoSQL	Yes	Yes	No	[48, 158]
Relationship models				
1-to-1	Yes	Yes	Yes	[48, 51]
1-to-Many	Yes	Yes	Yes	[48, 51]
Many-to-Many	Yes	Yes	Yes	[48, 51]
Inheritance	Yes	Yes	Yes	[48, 51]
Query patterns				
SQL	Yes	No	—	[51, 159]
Table Oriented	Yes	No	—	[160]
Document Oriented	No	Yes	—	[159]
CRUD	Yes	Yes	Yes	[48, 51]
Triggers	Yes	No	—	[51]
Cursors	Yes	Yes	Yes	[48, 51]
Notification	Yes	Yes	Yes	[48, 51]
Cascading Deletes	Yes	No	—	[51]
Scalability				
Automated Sharding	No	Yes	—	[50, 72]
Availability				
Replication	Yes	Yes	Yes	[48, 51]
Flexibility				
Modular	Yes	No	—	[51]
Stored Procedures	Yes	No	—	[51]
ACID-ity				
Referential Integrity	Yes	No	—	[51]
Transactions	Yes	Yes	Yes	[48, 49, 51]
ACID	Yes	Yes	Yes	[48, 49, 51]
Advanced features				
MapReduce	No	Yes	—	[48]
Database Federation	Yes	No	—	[51]

feature is listed as either “Yes” or “No” per database and whether the feature is equivalent in both database technologies. A column for references to the information is also provided.

On the relational side, given a traditional paradigm, a developer can depend on the database to maintain consistency, but scalability and flexibility become difficult and troublesome.

For NoSQL databases, a developer loses some built-in consistency and transactional capability, but scalability and flexibility are good. Due to the recent nature of the technology, some training may be required, which may impact a project's cost and duration. However, the added flexibility and code-oriented design may diminish the impact.

Hybrid approaches are use-case specific, and no strategy was found during the literature survey. Some work has been done to ease access to different database types, but no work has been done on the design process of hybrid data-based applications.

3.6 Conclusion

Figure 3.8 shows the overlap between systems engineering design and entity-relational design, as seen in the literature. There is an overlap between the preliminary design phase and the conceptual and logical modelling phases. However, the artefact of the systems approach is a *documented functional system*, including a database and application, whereas ER only delivers a *documented database*. The latter artefact then serves as the input to other functions, such as the application, reiterating the disconnect [23, 84].

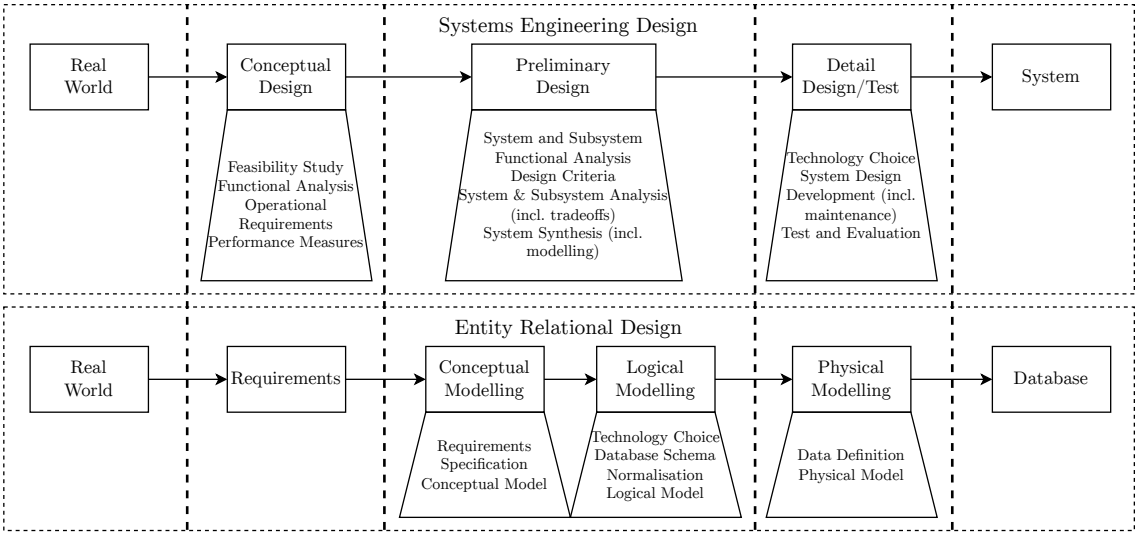


Figure 3.8: Systems Engineering [23] vs. ER Design Methodology [20] from [161, Fig. 2]

Since the most significant overlap occurs in the preliminary design phase, proper focus on this phase will greatly impact the system's overall performance. Furthermore, in SE, the preliminary design phase also includes a functional analysis, which incorporates other functional units and interfaces to them. These interfaces should also be considered when designing a database. Following this systems approach partly addresses the research context *(1) Process*.

A preliminary design, in the SE sense, is typically used to provide development specifications (Type B) for use in the detail design phase [23]. A development specification is critical as it creates a link between conceptual design and detail design; a link that is

often not present in data-based applications. In addition, the preliminary design focuses on modelling and evaluation of solution options, which must be done before detail design commences. One such option will be the choice of technology for a database solution, which can lead to success or failure when done incorrectly.

Selection of appropriate database technology, database structure, and other detail design considerations must be made before detail design commences. Therefore, it is necessary to understand the technology options and their relative strengths and weaknesses. As a result, experiments were defined to answer questions on the relative performance of two DBMSs in terms of clear performance metrics and the design parameters that will impact detail design.

Table 3.2 showed that the gap between the two databases is superficial when considering them at a high level. At a lower level, the performance characteristics of DBMSs must be determined with respect to changes in modelling parameters. The results of these experiments will determine (i) individual strengths and weaknesses in basic database operation performance and (ii) the modelling parameters that have the greatest impact on performance per database.

In terms of research management, it is necessary to summarise how concept research solutions were linked to specific research solutions for the sake of continuity. The summary is best presented in the RVM where higher-level concept solutions are linked to lower-level specific research solutions. The specific solutions show how each concept solution was addressed with experiments, use case analyses, conceptual modelling, preliminary modelling for relational and non-relational data models, critical evaluation, and the eventual design of a decision support system for hybrid applications. Table 3.3 shows the generic and specific research solutions.

Table 3.3: Research validation matrix with solutions

Concept / Research Solutions / Specific Research Solutions	Determine strengths and weaknesses of each database	Critically evaluate each model type and propose criteria	Evaluate different conceptual modelling techniques	Use best practices to model equivalently in both types	Use a full life cycle systems approach to data-based application design	Design a decision support framework for database design
Empirical database performance characterisation	×					
Identification of a common and representative database use case	×					
Conceptual modelling of application scenarios			×		×	
Modelling application scenarios for relational data model(s)			×	×	×	
Modelling application scenarios for non-relational data model(s)			×	×	×	
Critical evaluation of each model	×	×			×	
Design of a decision support framework for hybrid data-based applications		×		×	×	×

The true method of knowledge is experiment.

William Blake

Chapter 4

Characterisation

Introduction

Database performance is critical in the selection of an appropriate DBMS in the development process. The preliminary design depends on performance information to make an informed decision on database technology. In addition, modelling parameters are important for decision-making as these are both technology- and application-dependent.

This chapter describes experiments and results of empirical database experiments on the two databases from a practical perspective — that is, experiments were conducted in the form of tests. The results and findings from this chapter highlight which modelling parameters impact performance most. These modelling parameters form the basis for formulating proper database modelling and design criteria.

The results and findings presented in this chapter were validated partly by a publication in the 2022 SAIIEE33 conference proceedings [161].

4.1 Experimental Setup

CRUD operations form the basis of all simplistic and complex database operations. Therefore, a set of experiments covering a wide range of parameters applicable to the basic operations were chosen. The impact of each on the system's operational performance will be analyzed.

Typical parameters that were varied in iterations are:

1. Number of indexes[1];
2. Number of fields projected;

[1]The plural form *indexes* is used in preference to *indices*, as is the norm in database literature.

3. Number of fields in data;
4. Number of fields used in queries;
5. Number of threads used;
6. Limiting results returned and operated on;
7. Batch sizes.

The cost of transactions on empirical operations was evaluated in preliminary tests and was found to have no significant impact on the results. Transactions could impact complex operations but will not be considered for the empirical test suites.

Empirical experimentation requires a consistent environment with as slight as possible variation in conditions while keeping in mind the assumptions made by the choice of environment.

C++ was chosen as the programming language for writing the tests. Google's Test[2] and Benchmark[3] frameworks were used for experimentation and result tracking. All test suites were run three times, and each experiment was run for many iterations to establish statistical stability.

Tests were run on commodity hardware:

- Ubuntu 20.04 x86-64
- Intel® Core™ i7-4770 CPU @ 3.40GHz
- 32GiB RAM DDR3 1600 MHz
- Samsung EVO 850 SSD

Tests were rerun on two other machines to verify that the characteristic results and patterns were repeatable in similar environments.

- Ubuntu 20.04 x86-64
- Intel® Core™ i7-6500U CPU @ 3.10GHz
- 8 GiB RAM DDR3 1600 MHz
- Samsung EVO 650 SSD

and

[2] `https://github.com/google/googletest`
[3] `https://github.com/google/benchmark`

- MacBook Pro (13-inch, 2017, Two Thunderbolt 3 ports)
- 2.3 GHz Dual-Core Intel® Core™ i5
- 8 GiB 2133 MHz LPDDR3

MongoDB v4.4 with `libmongocxx v3.6.2` and PostgreSQL v13 with `libpqxx v7.3.0` were used for testing.

This research performed functional performance tests with which to characterize the performance of different DBMS technologies and their basic functionality of all databases (CRUD). These empirical tests were conducted outside of the cloud environment as these were relativistic comparisons that were not dependent (relativistically) on the machine infrastructure as such.

Source code and instructions are provided in Appendix D.

4.2 Create/Insert Experiments

4.2.1 Setup

The number of indexes is restricted by the number of fields in the iteration; for example, if the combination has only eight fields, the number of indexes is limited to 8 for the suite, as shown in Table 4.1.

Table 4.1: Parameter setup for inserts

Parameter	Minimum Value	Maximum Value	Step/Increment
Batch Size	1	4096	×8
Fields	1	16	×2
Indexes	0	16	×2
Threads	1	8	+2

Before each suite of tests, the database is cleared, and indexes (if any) are removed.

4.2.2 Test 1 - Insert

This test measures the impact of batch size, number of fields and indexes on insert performance.

4.2.3 Test 2 - Bulk Insert (MongoDB only)

MongoDB has a unique bulk-write operation. This test is based on Test 1 but utilises the *insert_one* bulk write operation. PostgreSQL does not offer an alternative like this for online operational bulk writing.

4.2.4 Results

Table 4.2: Summary of insert performance: Operations per second

Ops/Second	Minimum	Maximum	Average
MongoDB	3459	942573 (699730)	162671
PostgreSQL	159	310147	54109

Table 4.2 shows the minimum, maximum and average operations per second for inserts. MongoDB has an additional mode that has delivered higher performance reflected in the table.

From Figure 4.1 both databases exhibit the same behaviour when varying the three parameters. Performance increases logarithmically with a more significant number of documents. In contrast, adding more fields or indexes attenuates performance logarithmically.

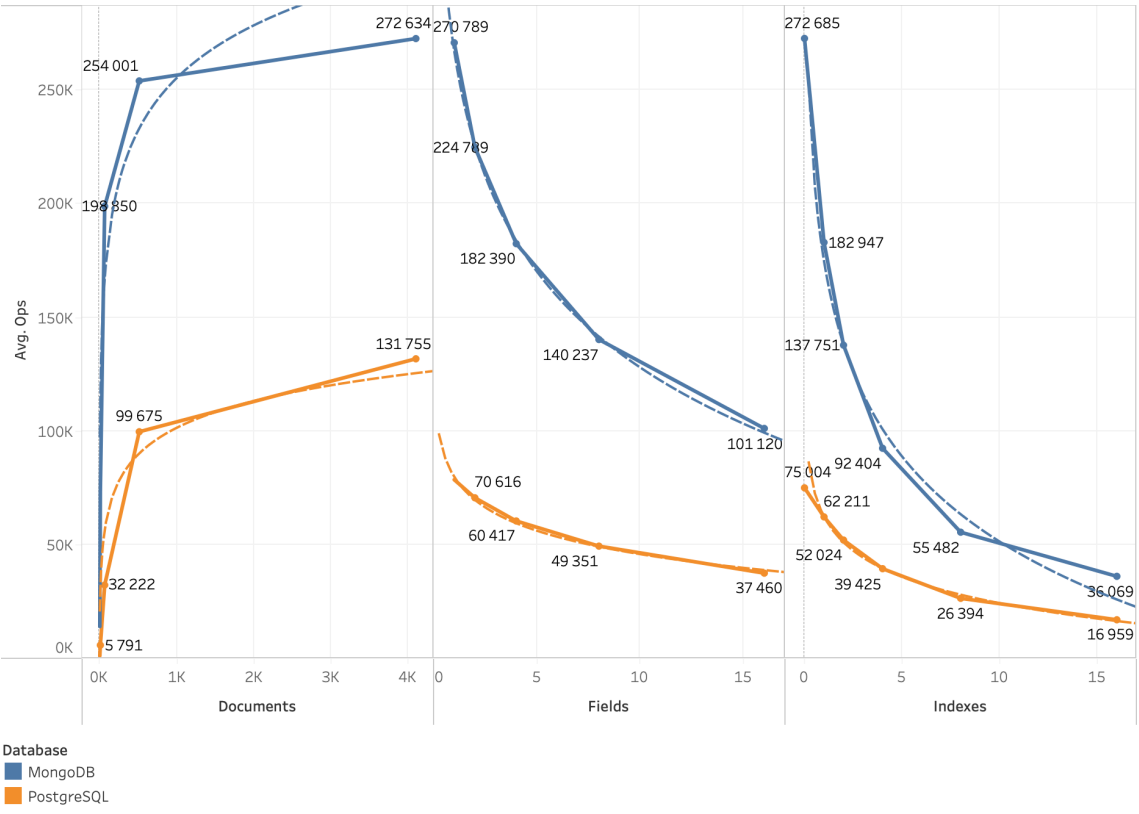


Figure 4.1: Average Ops/Second for inserts

MongoDB performs 36% better with the bulk insert method than with the regular insert

operation. This improvement could be ascribed to the fact that there is less overhead to process when sending in bulk. Figure 4.2 shows the difference.

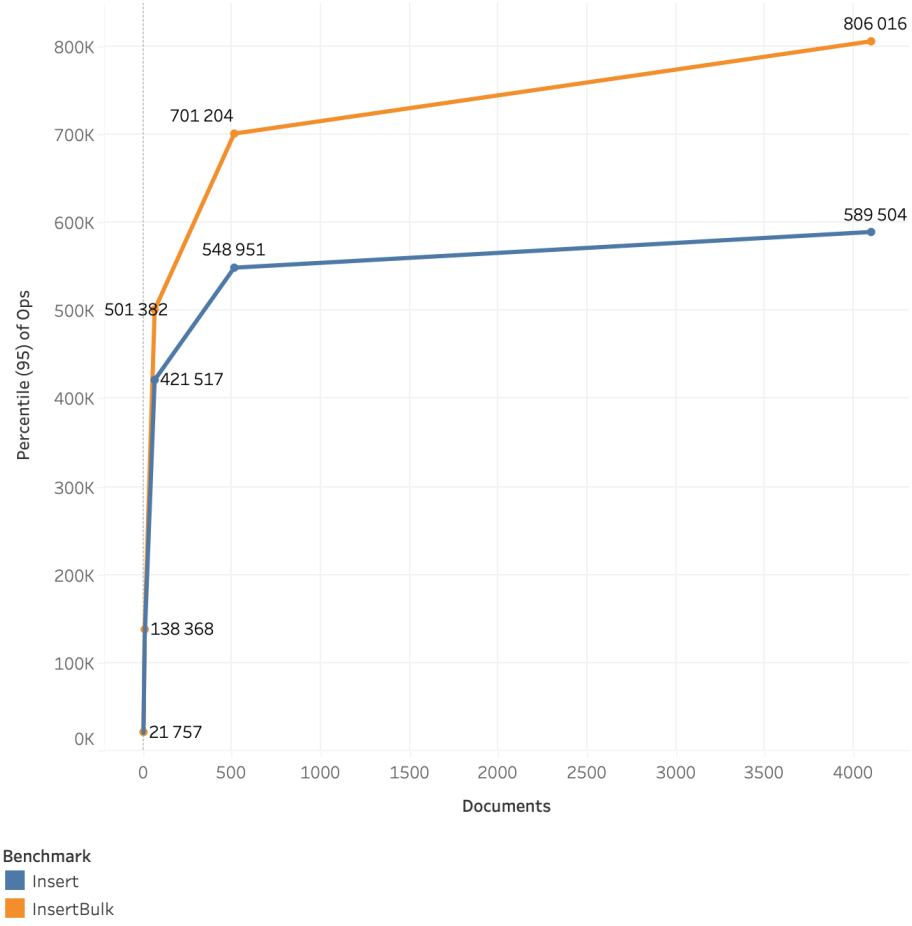


Figure 4.2: Insert bulk vs normal insert

Figure 4.3 shows the %-difference when adding more threads, with 1-thread as reference. Threads have a more significant advantage for MongoDB, with both databases exhibiting a declining logarithmic behaviour. PostgreSQL displays signs of saturation near the 8-thread mark, indicating that other bottlenecks have been reached, such as disk performance. Clearly, the workload is not CPU-bound since it still improves performance past the actual physical CPU limit.

Field impact, as shown in Figure 4.1, includes the impact of indexes in the results average calculation. Therefore, to indicate the field-only effect, Figure 4.4 shows the average operations per second on test iterations with no secondary indexes. Here it can be seen that a nearly linear decline is apparent. MongoDB is slightly more affected, with a roughly 50% drop in performance for every 16 fields added compared to the 36% drop of PostgreSQL.

PostgreSQL's performance is greatly improved by adding more documents at a time. It can be seen from Figure 4.5 that when inserting 4096 documents per operation, the increase in operations per second is 11834% compared to single inserts for PostgreSQL, compared to the 1847% increase for MongoDB. MongoDB's baseline performance, however, still results in more operations per second.

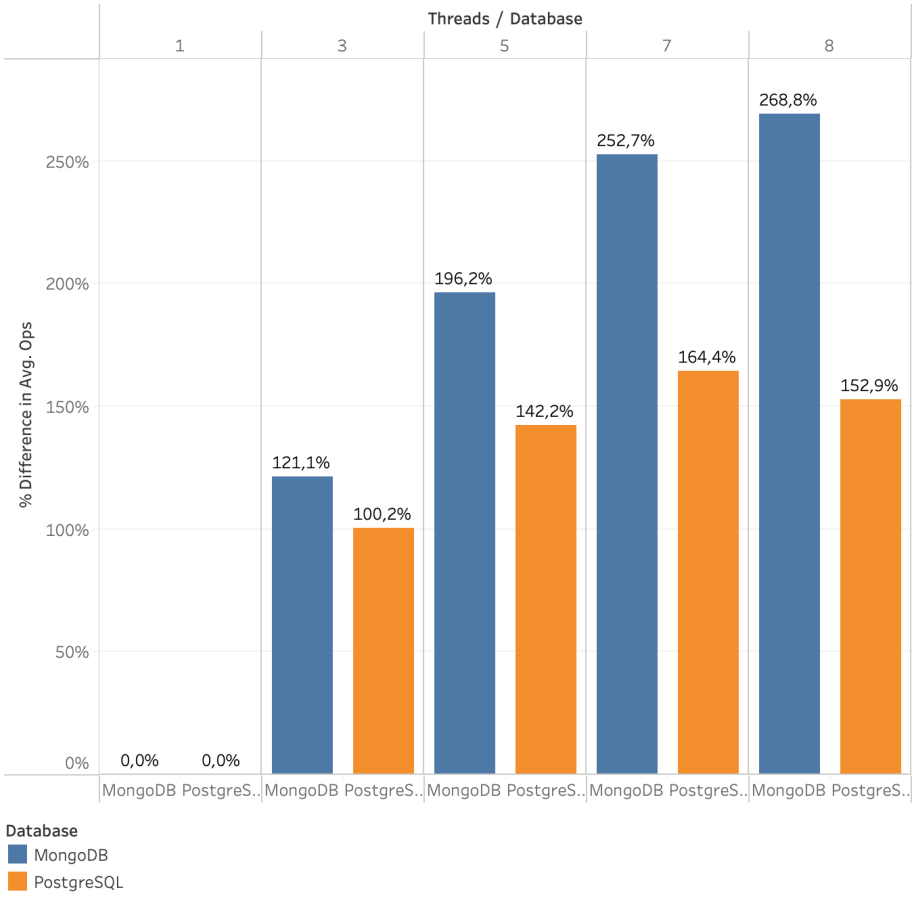


Figure 4.3: Thread impact for inserts

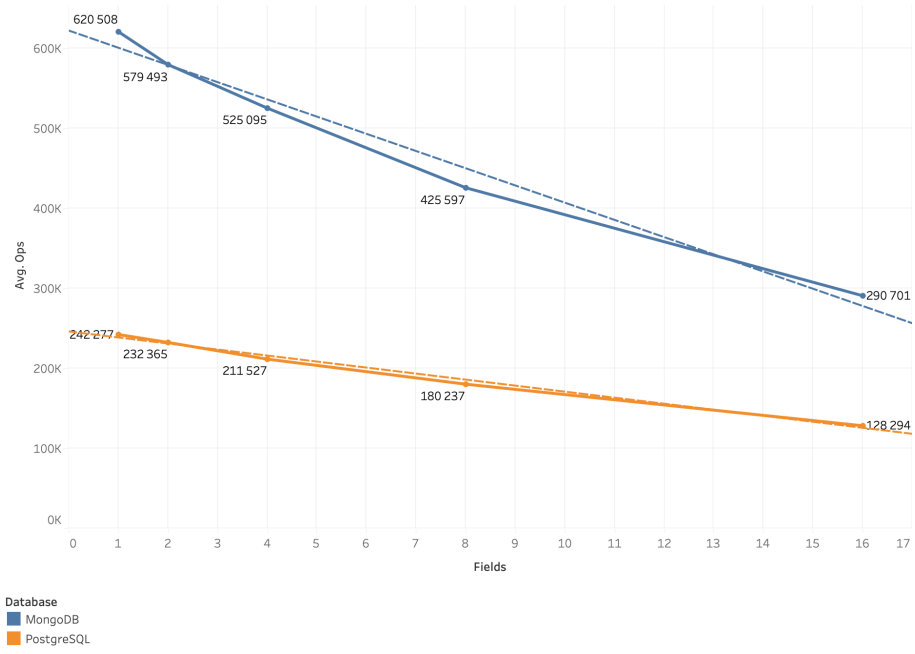


Figure 4.4: Field-only impact for inserts

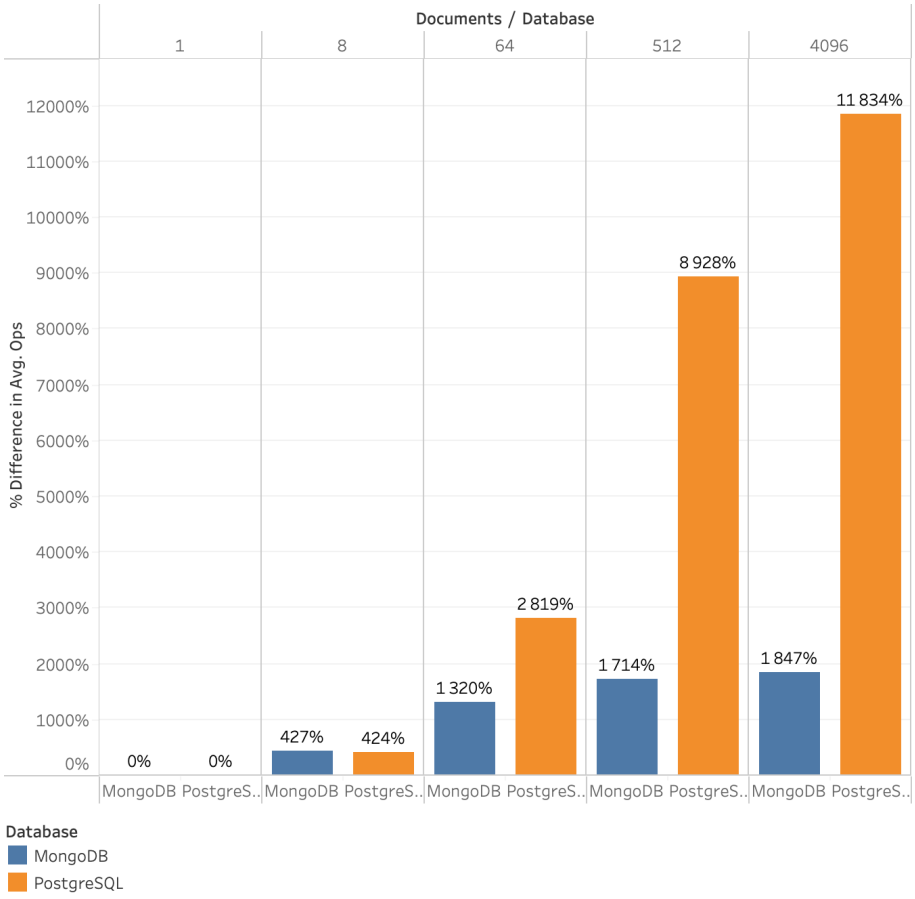


Figure 4.5: Document impact for inserts

From these results, the impact of each parameter closely follows the same pattern for both MongoDB and PostgreSQL. MongoDB is roughly 30% faster with inserts on nearly all the experiments. Furthermore, Figures 4.1 and 4.4 show that the results are predictable, as seen by the fit curves illustrated with dotted lines.

4.3 Read/Query Experiments

4.3.1 Setup

Read or query operations do not cause any changes to the database. Therefore for this suite of tests, a database had been pre-populated with 1 000 000 documents/rows, each with six fields with values ranging from one to ten. Each permutation of the ten values in 6 fields is present in the 1 000 000 documents/rows.

Depending on the index parameter, the indexes are removed/or added for each set of parameters. The tests have been divided into three categories:

1. Aggregations: Average, multiplication, sum and count;

2. Standard queries: finding and sorting;

3. Joins: self-joins and manual self-joins.

Where applicable, the limit axes are shown on a logarithmic scale to demonstrate the behaviour more clearly. Where applicable, the limit was clamped to the maximum possible for the selected parameter set.

4.3.2 Aggregations

The only parameters used in these experiments are the limit of how many rows/documents to return and the number of threads used to query the databases, as shown in Table 4.3. The Count aggregation experiment only varies the number of fields queried and the number of indexes, as listed in Table 4.4.

Table 4.3: Parameter setup for aggregations

Parameter	Minimum Value	Maximum Value	Step/Increment
Limit	1	524288	×2
Threads	1	8	+2

Table 4.4: Parameter setup for Count aggregation

Parameter	Minimum Value	Maximum Value	Step/Increment
Fields	0	6	+1
Indexes	0	6	+1

The results are shown in Figure 4.6 on a logarithmic x-axis. Both databases exhibit the same pattern for increasing the limit. It is important to note that whilst operations per second diminish with larger limits, the number of rows/documents per second is inversely proportional.

Figure 4.7 shows a curve fitting in the saturated region which is a natural power function. The unsaturated region is shown in Figure 4.8. This region is fitted well with a negative exponential function. The region to the left, the so-called unsaturated region, is due to the database doing little batched work but also not scanning many documents.

The saturated region is where the database needs to scan more documents and return more, adding overhead and processing.

The sum aggregation exhibits the same behaviour, which is expected since the database scans the same number of elements but applies another operator. The results are shown in Figure 4.9.

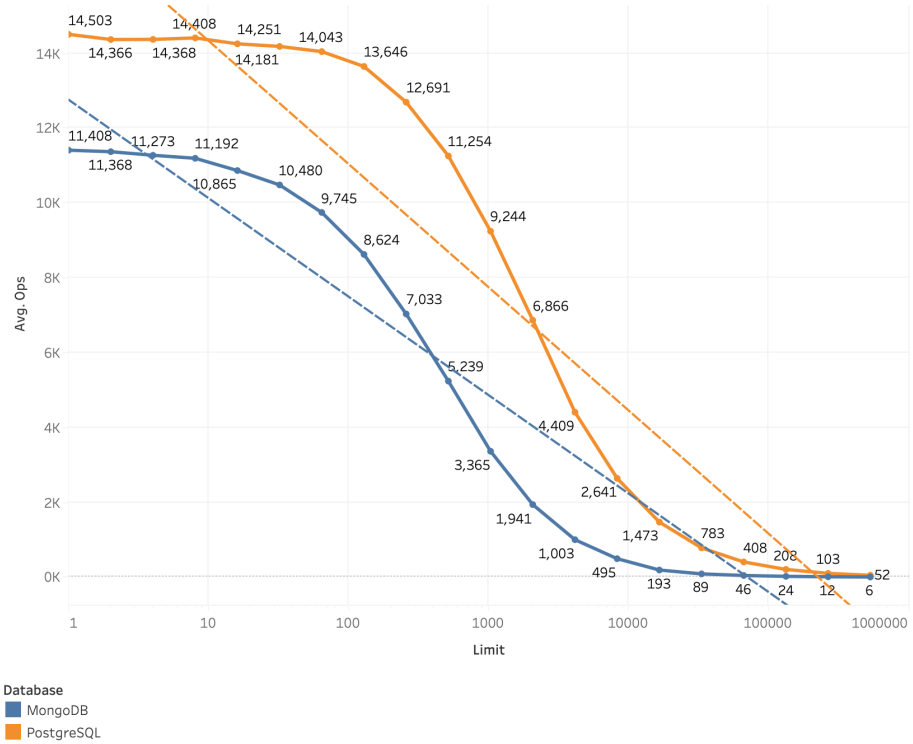


Figure 4.6: Average Ops/Sec for Average aggregation

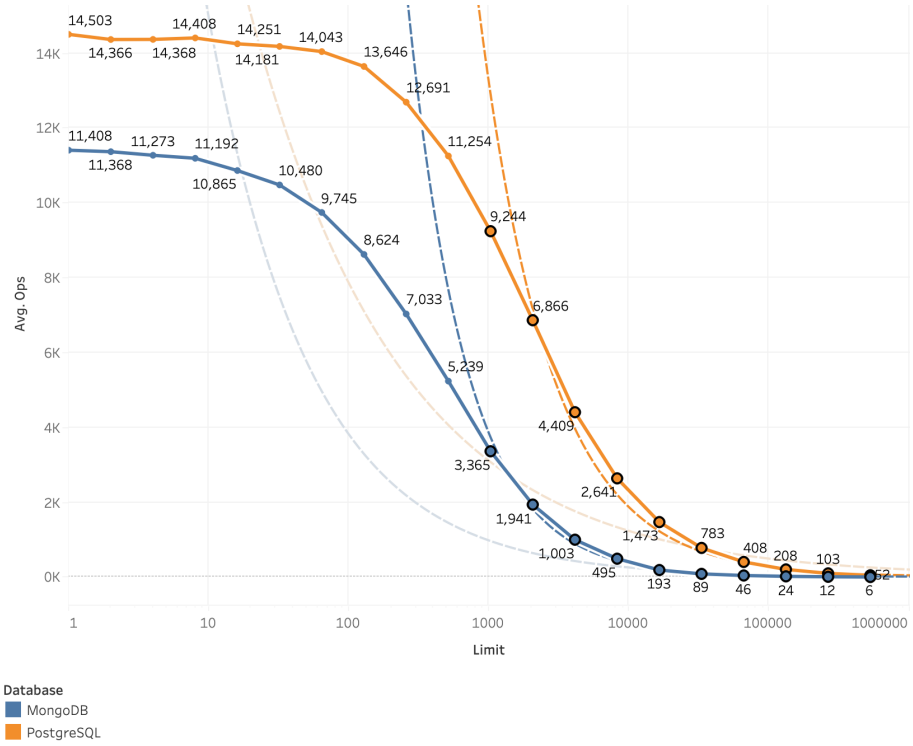


Figure 4.7: Saturated region for Average aggregation

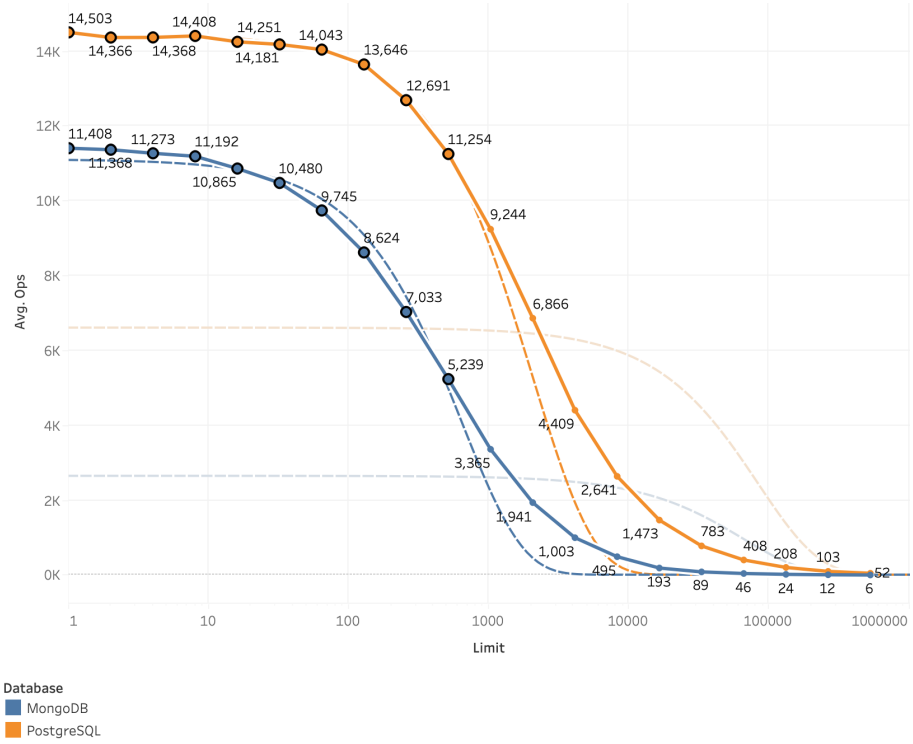


Figure 4.8: Unsaturated region for Average aggregation

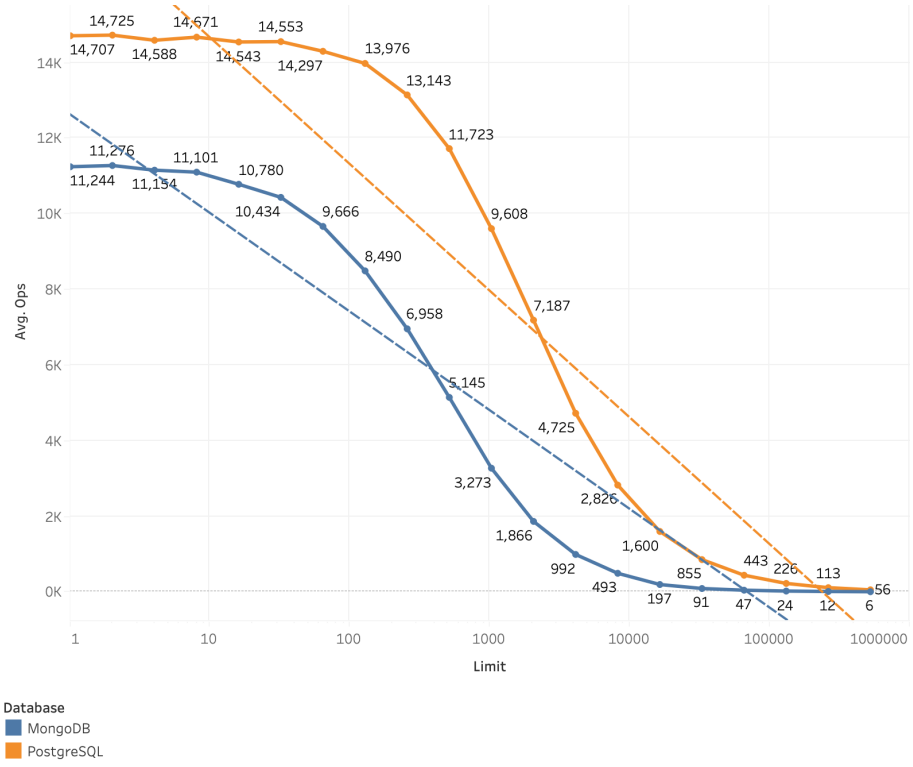


Figure 4.9: Average Ops/Sec for Sum aggregation

Figure 4.10 shows the results for the multiplication operation. Again, this exhibits the same behaviour as the previous aggregations. Here a power function is fitted to show that it also works well, especially in the saturated region. It is interesting to note that the average ops/second is close in all three aggregation tests.

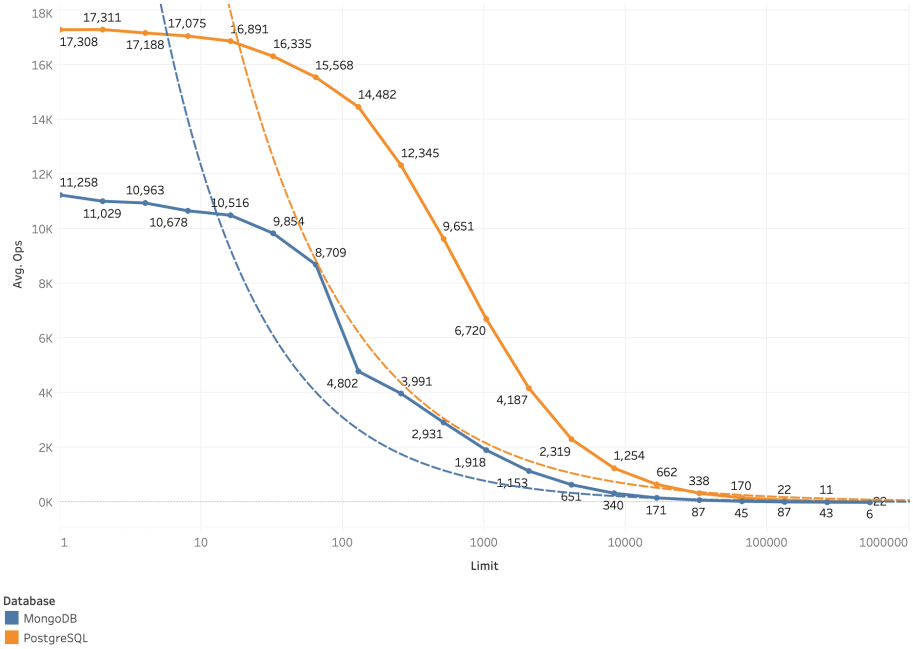


Figure 4.10: Average Ops/Sec for Mul aggregation

The count aggregation has a different impact on the database since a limit was not applied, and the implication is that the database needs to scan the whole table or collection. The results are given in Figure 4.11, and Figure 4.12 shows the cases with no indexes applied.

Both databases improve with additional threads very close to each other as expressed in Figure 4.13.

Once again, both databases show the same trends and the constant trend when no indexes are applied is seen. This behaviour is because the whole table/collection must be scanned when no indexes are present.

Only the index needs to be scanned with indexing, resulting in exponential gains. The more fields that are fixed (and indexed), the fewer elements must be processed and searched, leading to more operations per second.

4.3.3 Standard Queries

Standard queries have been run on the parameter set shown in Table 4.5. Figures 4.14 and 4.15 show the average operations per second for unsorted and sorted general queries, respectively.

Figure 4.16 shows the performance of unindexed sorted queries. It can be seen that there

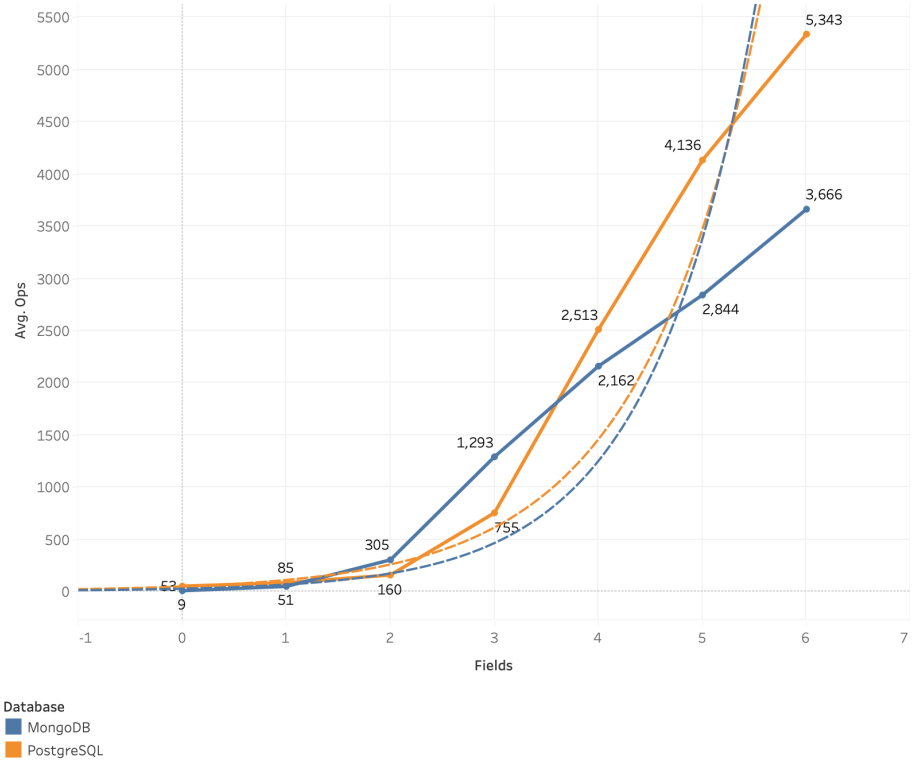


Figure 4.11: Average Ops/Sec for Count aggregation

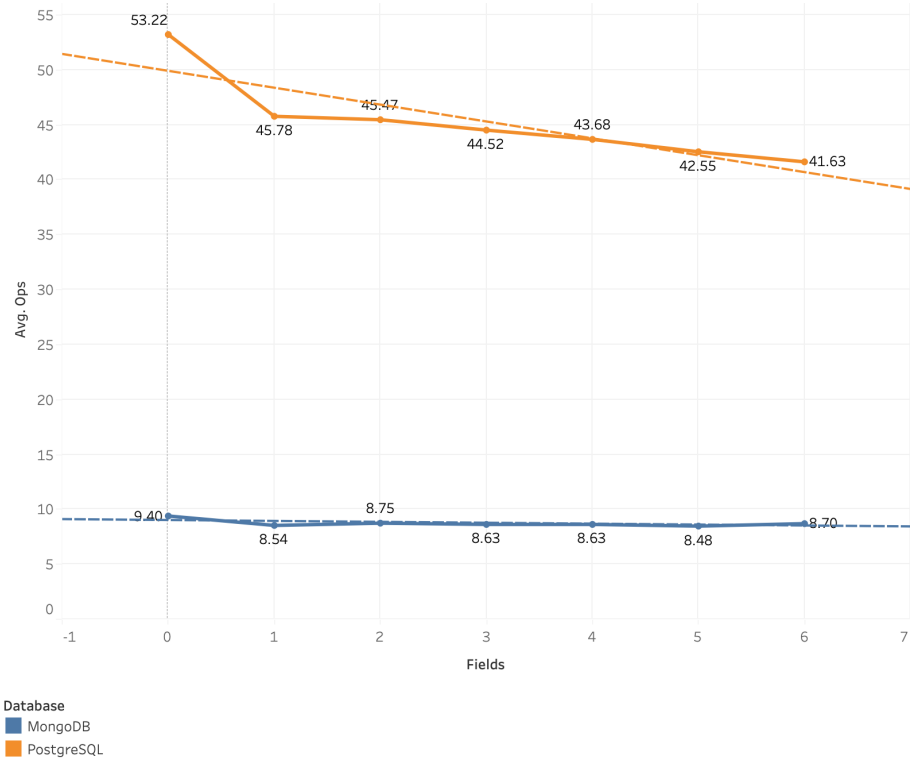


Figure 4.12: Average Ops/Sec for Count aggregation (no indexing)

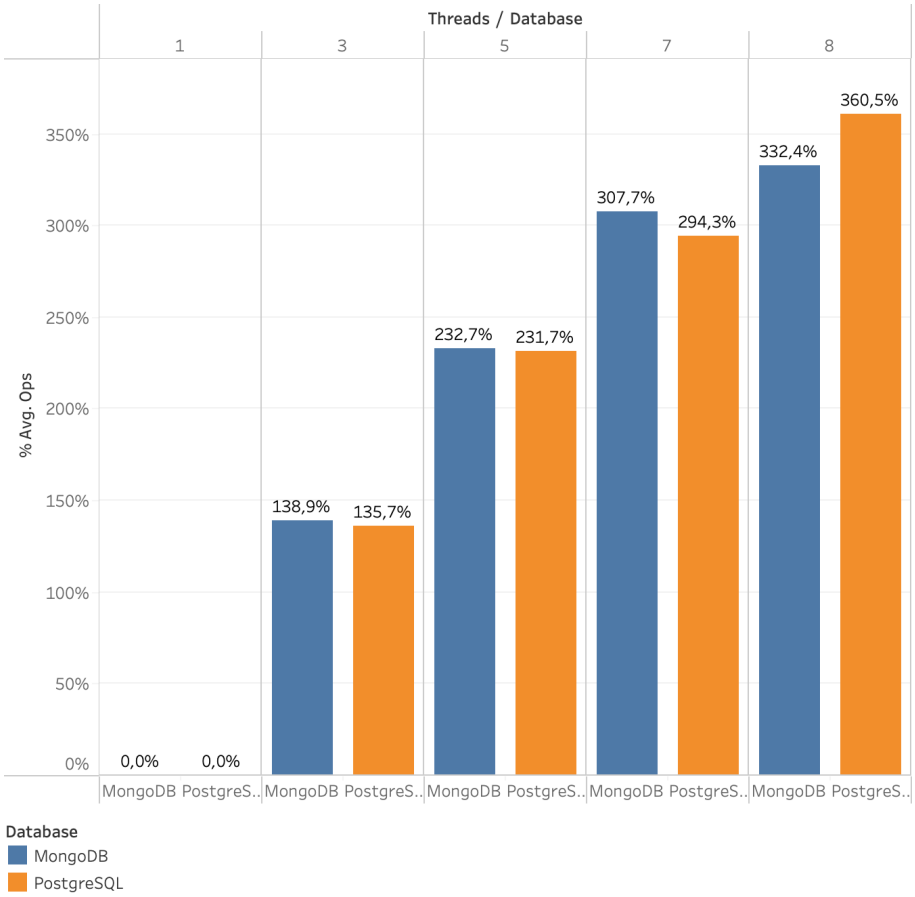


Figure 4.13: Impact of threads on aggregations

Table 4.5: Parameter setup for standard queries

Parameter	Minimum Value	Maximum Value	Step/Increment
Fields	0	6	+1
Fields Returned	0	6	+1
Indexes	0	6	+1
Limit	1	1000000	×2
Threads	1	8	+2

is an anomaly in the PostgreSQL results, while MongoDB performs predictably, as shown by the fitted exponential function.

The anomaly seen in PostgreSQL's boosted performance for a limit less than 20165 can be ascribed to PostgreSQL's work planner. For this specific setup, queries for limits less than 20165 used the top-N heapsort in-memory algorithm. For any limit above 20165, PostgreSQL used the external merge disk-based method, and the latter performs much slower due to the disk access required [51]. Figure 4.17 shows the fit for the lower section.

Listing 4.1 shows the `psql` session showing the two sort methods used under different limits.

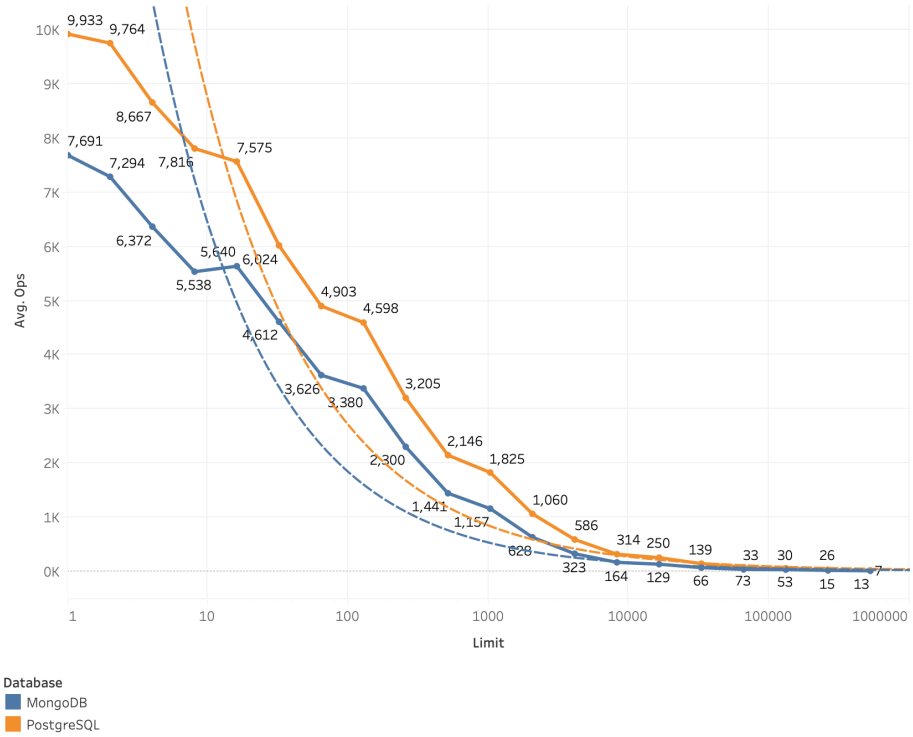


Figure 4.14: Average Ops/Sec for general queries

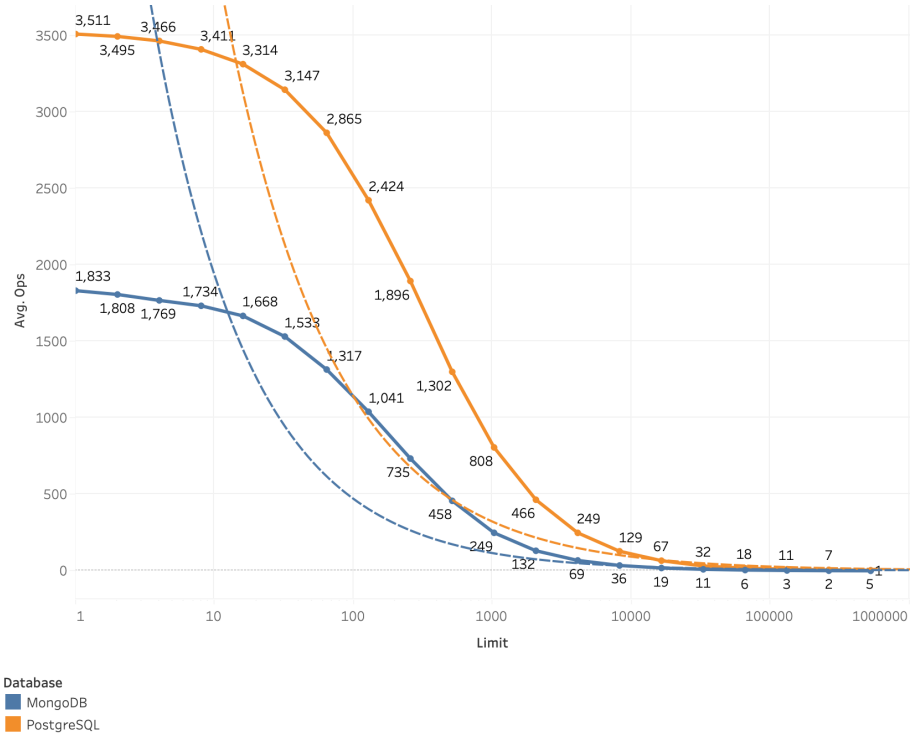


Figure 4.15: Average Ops/Sec for sorted queries

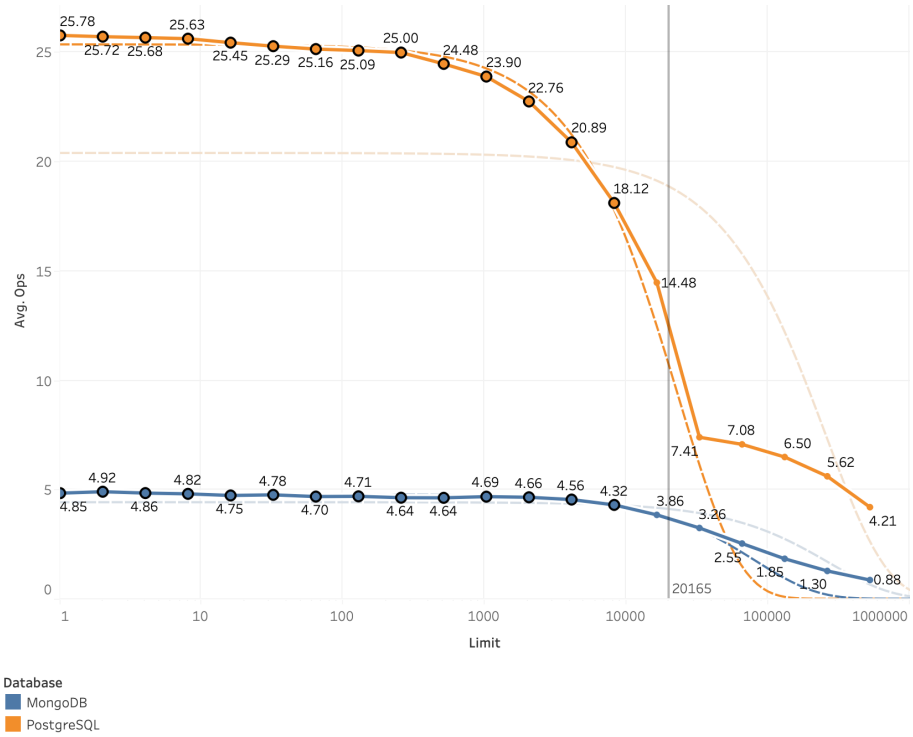


Figure 4.16: Average Ops/Sec for unindexed sorted queries

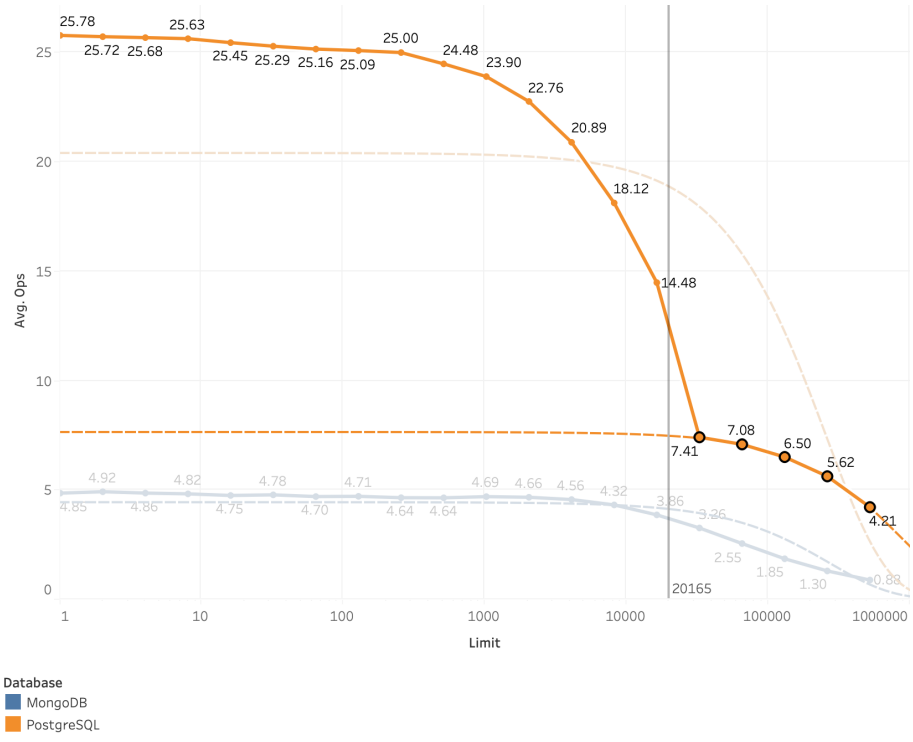


Figure 4.17: Average Ops/Sec for unindexed sorted queries cont.

```
pieter=# EXPLAIN ANALYSE SELECT _id  from bench.read_bench ORDER BY a0 DESC,a1 DESC,a2 DESC,a3 DESC,a4 DESC,a5 DESC LIMIT 20165;
                                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=44393.81..46746.56 rows=20165 width=28) (actual time=293.314..307.099 rows=20165 loops=1)
   ->  Gather Merge  (cost=44393.81..141622.90 rows=833334 width=28) (actual time=293.312..306.219 rows=20165 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Sort  (cost=43393.79..44435.46 rows=416667 width=28) (actual time=284.749..286.967 rows=7363 loops=3)
               Sort Key: a0 DESC, a1 DESC, a2 DESC, a3 DESC, a4 DESC, a5 DESC
               Sort Method: external merge  Disk: 11608kB
               Worker 0:  Sort Method: external merge  Disk: 12592kB
               Worker 1:  Sort Method: external merge  Disk: 13032kB
               ->  Parallel Seq Scan on read_bench  (cost=0.00..11519.67 rows=416667 width=28) (actual time=0.009..19.380
↪  rows=333333 loops=3)
 Planning Time: 0.081 ms
 Execution Time: 309.699 ms
(12 rows)

pieter=# EXPLAIN ANALYSE SELECT _id  from bench.read_bench ORDER BY a0 DESC,a1 DESC,a2 DESC,a3 DESC,a4 DESC,a5 DESC LIMIT 20164;
                                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=44393.66..46746.29 rows=20164 width=28) (actual time=117.726..129.771 rows=20164 loops=1)
   ->  Gather Merge  (cost=44393.66..141622.75 rows=833334 width=28) (actual time=117.725..128.904 rows=20164 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Sort  (cost=43393.64..44435.31 rows=416667 width=28) (actual time=114.485..115.082 rows=7362 loops=3)
               Sort Key: a0 DESC, a1 DESC, a2 DESC, a3 DESC, a4 DESC, a5 DESC
               Sort Method: top-N heapsort  Memory: 3612kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 3614kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 3613kB
               ->  Parallel Seq Scan on read_bench  (cost=0.00..11519.67 rows=416667 width=28) (actual time=0.009..20.639
↪  rows=333333 loops=3)
 Planning Time: 0.085 ms
 Execution Time: 130.494 ms
(12 rows)
```

Listing 4.1: PostgreSQL explain output for unindexed sorting

The impact of threads is shown in Figure 4.18. Both databases scale well with threads, even beyond the physical CPU cores, reiterating that the workload is disk-bound.

Indexing greatly improves the average operations per second above 25000% for MongoDB and above 29000% for PostgreSQL. Indexes offer the DBMS the ability to fast-track to relevant data, exactly as a book's index does. Only the index must be scanned to get the correct rows/documents, depending on how many fields are queried and how many are indexed. Figure 4.19 shows the impacts.

This impact is especially true for sorted queries, as shown in Figure 4.20. MongoDB experiences the most significant gains when using indexes, reaching above 82000% increase over no indexes, and PostgreSQL achieves over 33000% improvement, close to its unsorted queries. Both, however, exhibit the same order of magnitude in this respect.

MongoDB is more negatively impacted by the number of fields queried when no indexes are present. Both databases reach critical mass when 5 or 6 fields are queried, as seen in Figure 4.21. Depending on the limit, the impact is exaggerated. The more documents/rows to return, the more scanning is required at the table/collection level.

Adding fields in the query when indexes are present has less of a disadvantage, where peak attenuation happens on four fields. This behaviour is shown in Figure 4.22. The

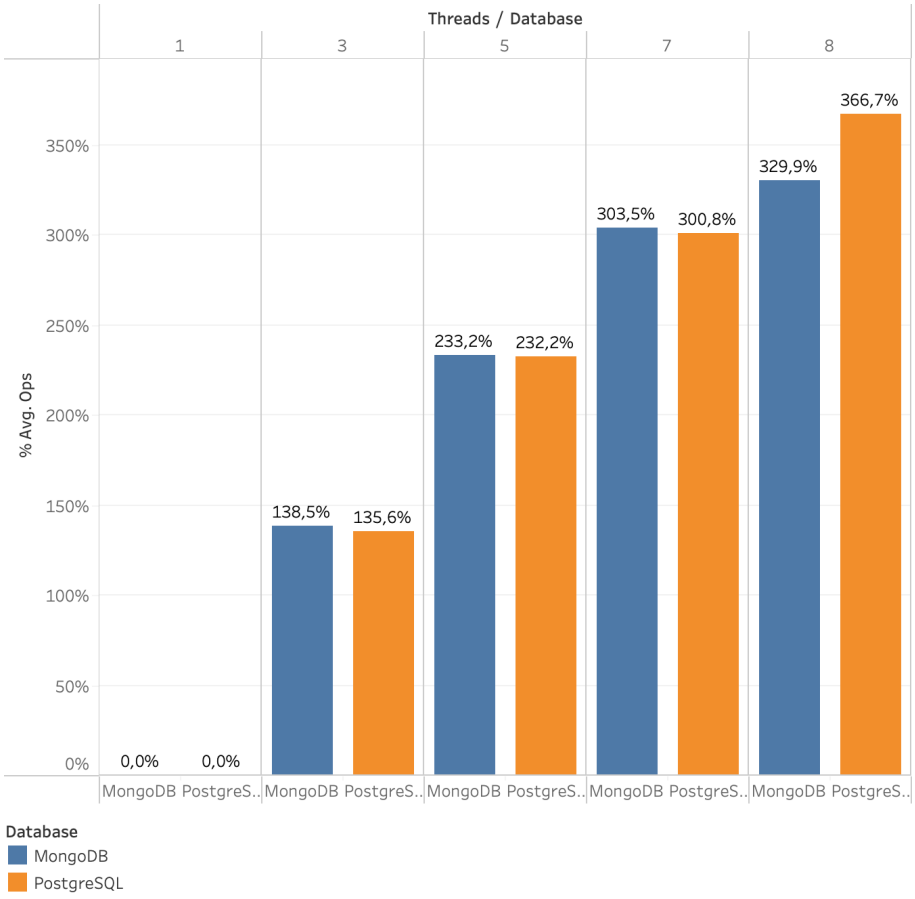


Figure 4.18: Impact of threads on queries

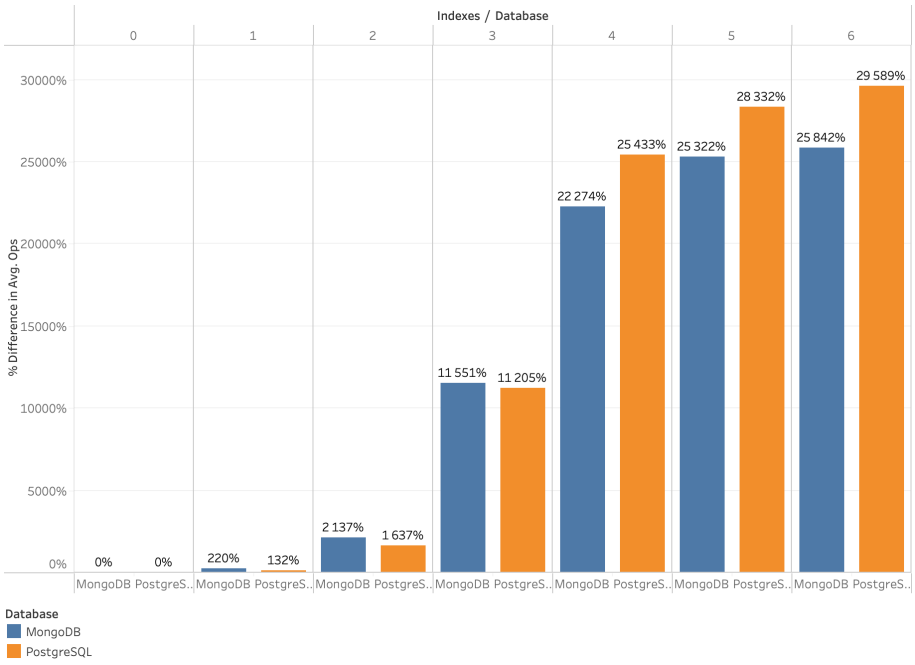


Figure 4.19: Impact of indexes on queries

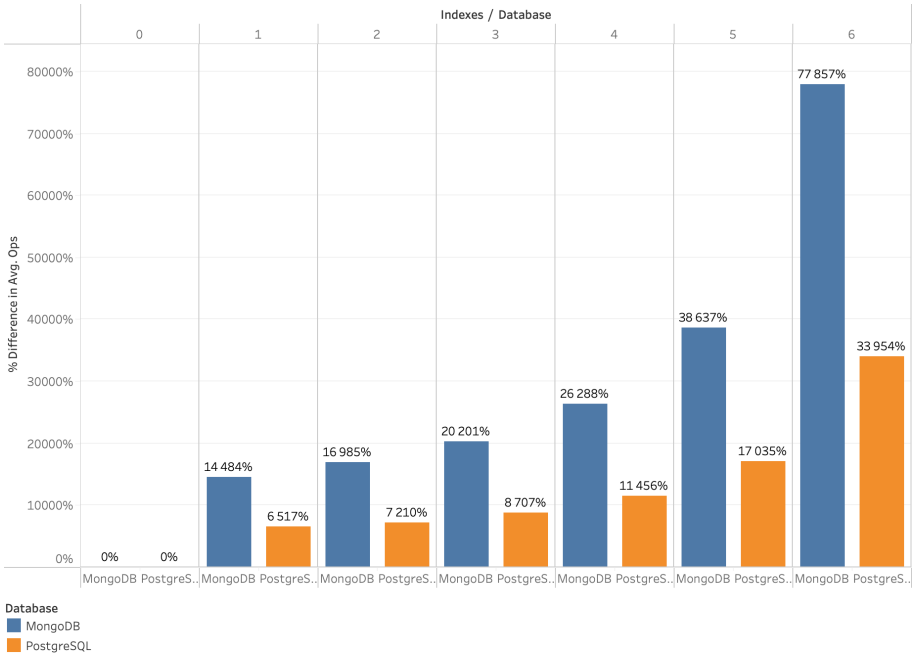


Figure 4.20: Impact of indexes on sorted queries

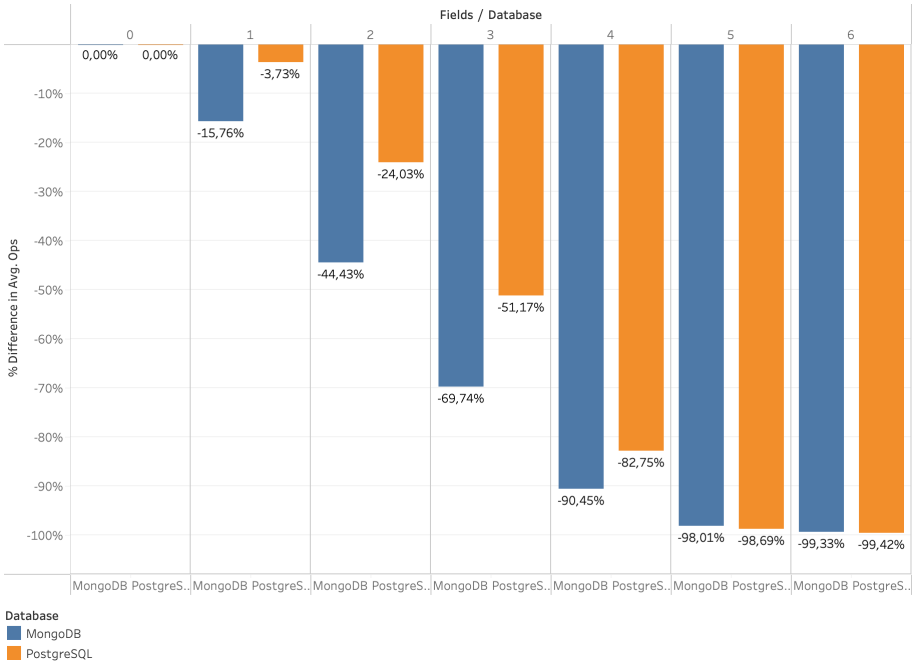


Figure 4.21: Impact of fields on queries (no indexes)

bell shape around four fields shows that adding more fields to the query limits the number of rows/documents returned by the operation, even for differing limits. For example, six fields fixed will only return one row/document. It is important to note that introducing indexes also diminishes the effect that fields have on query performance.

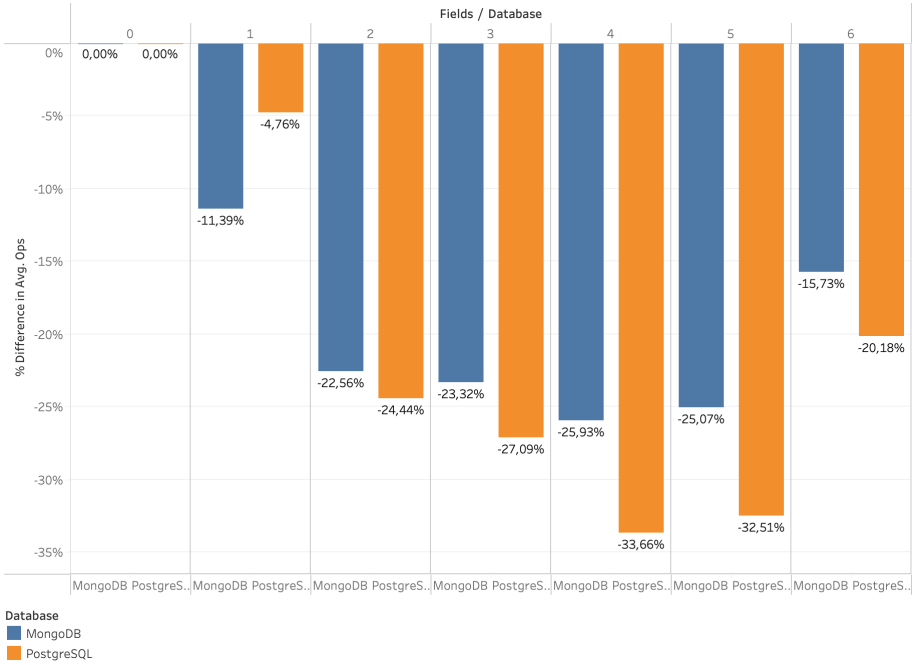


Figure 4.22: Impact of fields on queries

In Figure 4.23, it can be seen that the addition of fields decreases performance. There is one more anomaly with PostgreSQL for cases where six fields are queried, and PostgreSQL does not perform sorting when it only gathers one row, while MongoDB seemingly does not use this optimization.

Expected behaviour can be seen in Figure 4.24, where both databases closely follow the same behaviour. It can be seen that the more fields are queried, the worse the performance becomes.

Another factor to consider is the impact of fields projected, that is, fields that are presented after the query. Figure 4.25 shows the effect for both database systems. The more fields are projected, the more data must be transferred over the network and between disk and memory. The average impact in this experiment is seemingly low. However, with higher limits, the effect is more significant, as shown in Figure 4.26 where the limit has been fixed to 524288.

4.3.4 Join and Manual Join

Joins are synonymous with SQL databases but not uncommon in NoSQL databases. Self-joins are a subset of join operations where the table/collection is joined. An inner join is used in these experiments. The parameter setups for the built-in and manual join

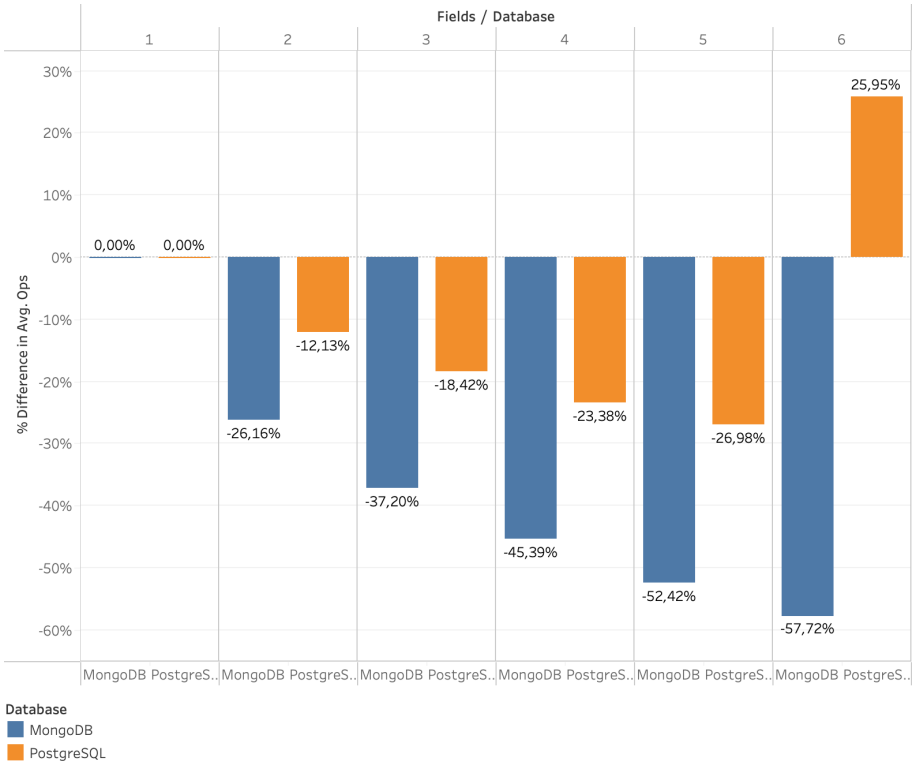


Figure 4.23: Impact of fields on sorted queries (no indexes)

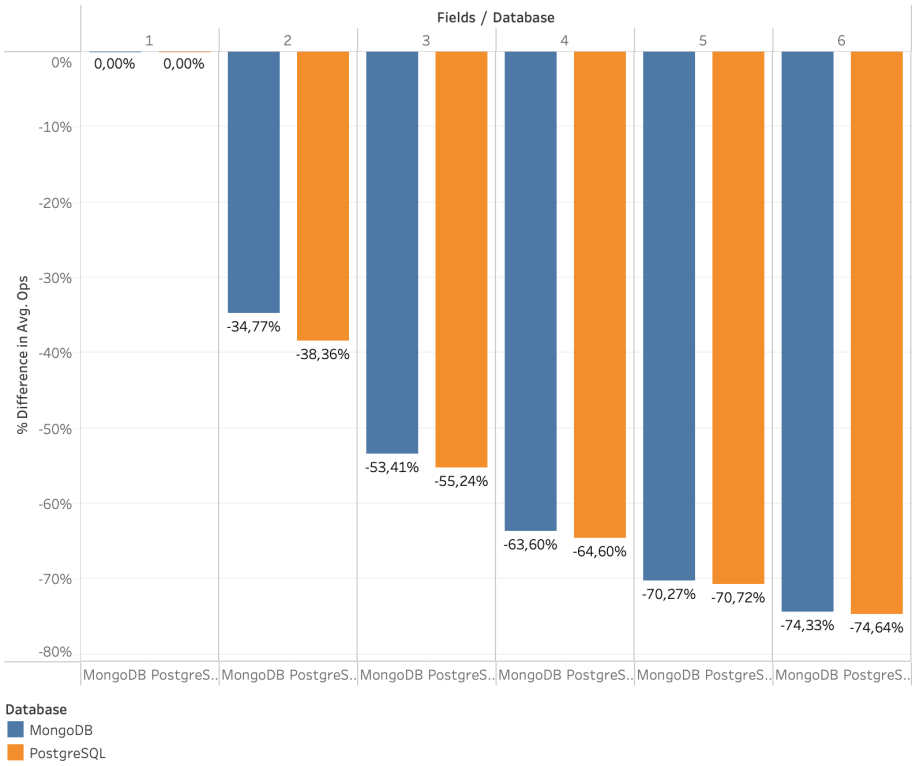


Figure 4.24: Impact of fields on sorted queries

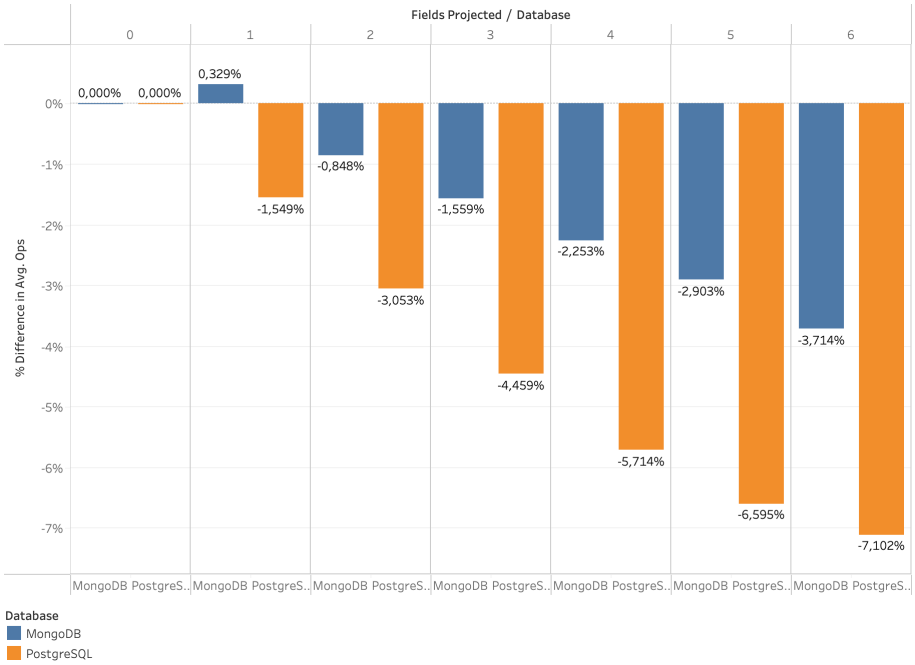


Figure 4.25: Impact of fields projected on queries

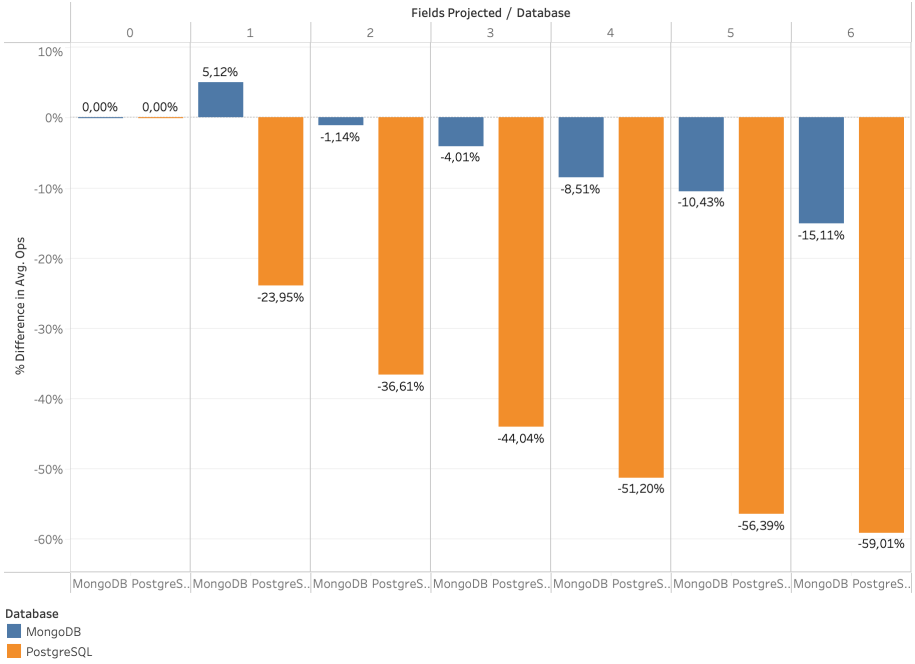


Figure 4.26: Impact of fields projected on queries, with high limit

experiments are shown in Table 4.6 and 4.7 respectively.

Table 4.6: Parameter setup for self-join queries

Parameter	Minimum Value	Maximum Value	Step/Increment
Indexes	0	2	+1
Limit	1	1000000	×2
Threads	1	8	+2

Table 4.7: Parameter setup for self-join manual queries

Parameter	Minimum Value	Maximum Value	Step/Increment
Indexes	0	2	+1
Limit	1	1000000	×2
Batch	1	4096	×2
Threads	1	8	+2

The performance of self-joins is shown in 4.27. It can be seen that PostgreSQL performs join operations extremely fast, and the behaviour closely resembles what was seen in previous read experiments.

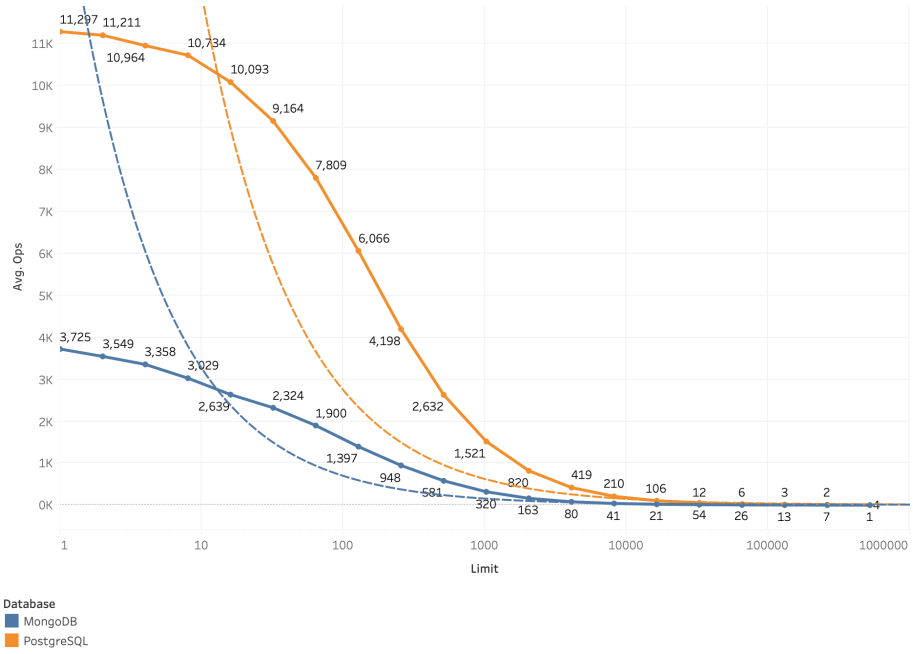


Figure 4.27: Average Ops/Sec for self-join

The experiment was repeated with a manual self-join, which means the initial query's results are used to form additional queries to the database from the application. This is in contrast to DBMS's built-in join functionality, where the joining is performed at the database level before sending the final results to the application. The results are shown in Figure 4.28.

In this case, MongoDB has an anomaly. The anomalous gain in performance can be

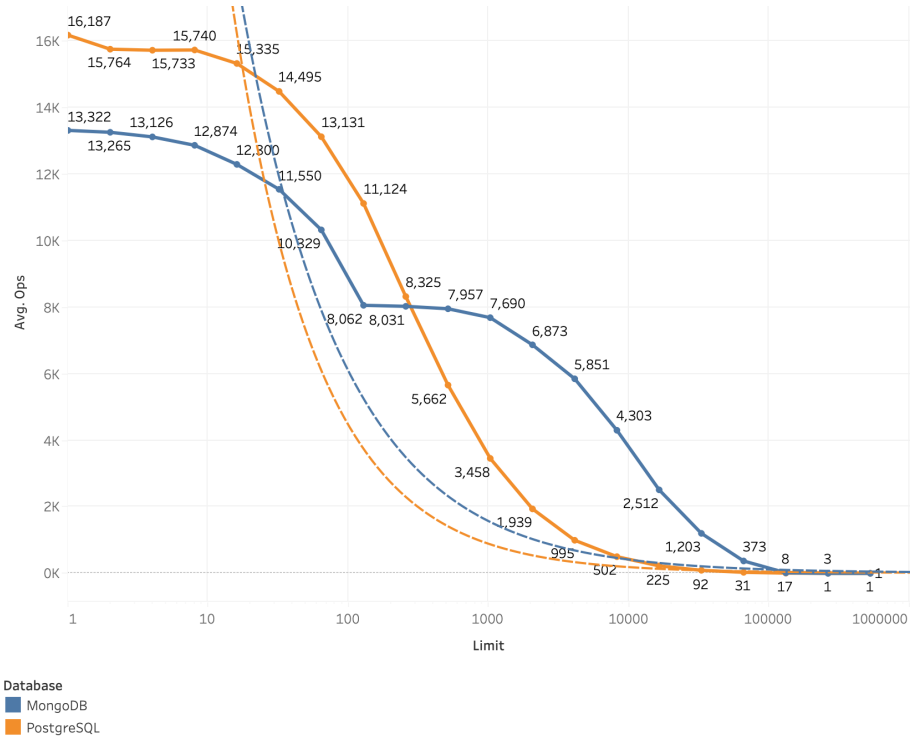


Figure 4.28: Average Ops/Sec for manual self-join

attributed to MongoDB's batching performance, as shown in Figure 4.29. It is interesting to note that PostgreSQL is very linear to batch size, whereas MongoDB has a significant performance gain for most batch sizes. It does, however, begin to saturate around 100.

The reason that batching makes such a significant impact is that the gains are exaggerated during higher limits.

Indexes in these join experiments have little effect because each row is self-joined. As demonstrated in other experiments, if queries were added to the join operations, it would have made a significant impact. The impact of indexes for joins and manual joins are shown in Figures 4.30 and 4.31 respectively.

The impact of threads is shown in Figure 4.32. As can be expected, threads make a significant improvement on both databases.

4.3.5 Summary

From all the experiments, it can be summarized that PostgreSQL outperforms MongoDB with reads. Both databases follow the same behaviour with changing parameters, with explainable anomalies described.

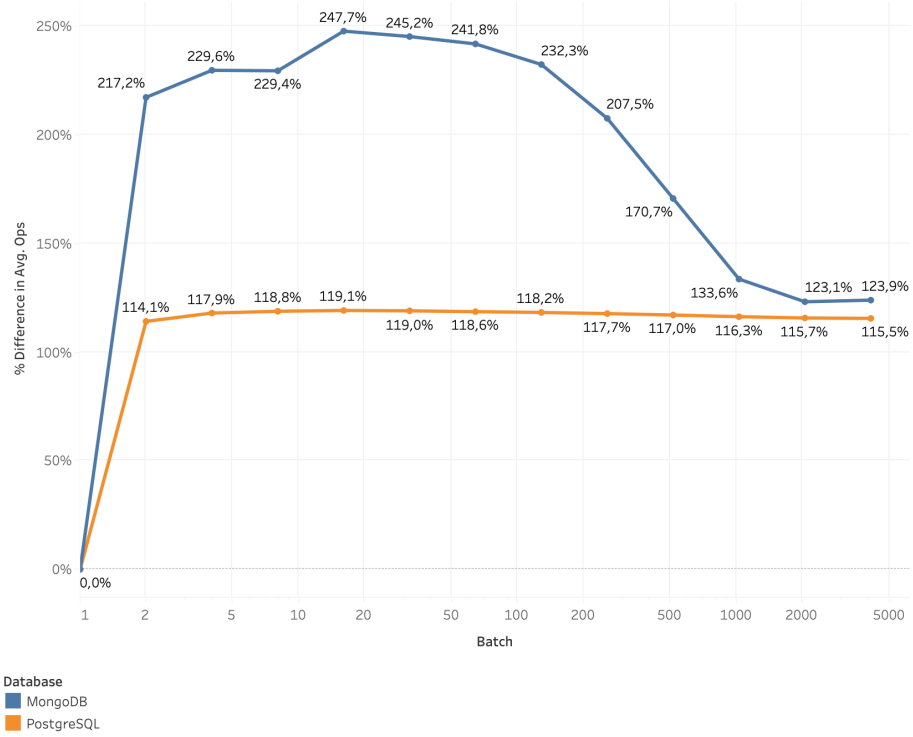


Figure 4.29: Impact of batch size for manual self-join

4.4 Update Experiments

4.4.1 Setup

Update operations have a query and a write component. In these experiments, no alterations to the schema are made during the operations, as this is not typically part of the operational processes in a production DBMS. Schema changes usually occur during maintenance processes and will be disregarded in these experiments.

The operation applied in the update operations is a simple random increment between 1 and 100. These experiments are again run on a pre-populated database, precisely like the read experiments. In this case, there are two sets of fields—one set for querying and one set for writing. Tests add and remove indexes as required by the parameters. For updating, measuring the impact for both read and write indexes is vital.

The following parameters are used and listed in Table 4.8. The write indexes are not always present, depending on the test suite.

4.4.2 Results

The first set of results shown in Figure 4.33 shows the impact of fields changed and fields queried when only considering read indexes. It can be seen that the fields changed has almost no impact on the performance of PostgreSQL and very slightly on MongoDB -

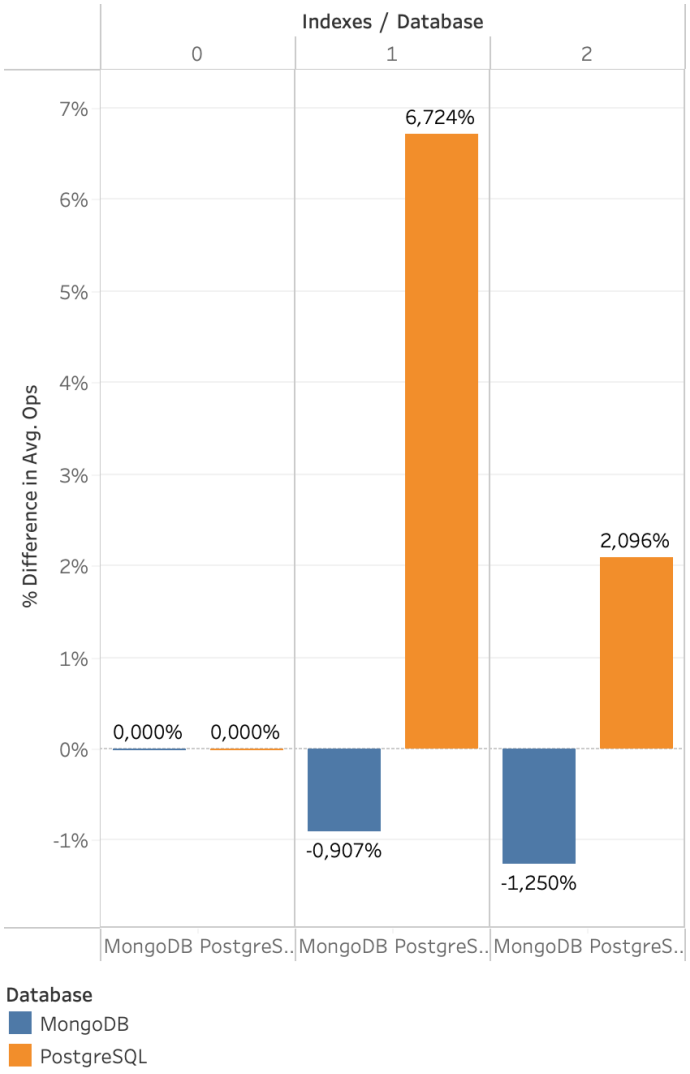


Figure 4.30: Impact of indexes for self-join

Table 4.8: Parameter setup for updates

Parameter	Minimum Value	Maximum Value	Step/Increment
Fields to query	0	6	+1
Fields to write	1	6	+1
Indexes for writing	0	16	+1
Indexes for reading	0	16	+1
Threads	1	8	+2

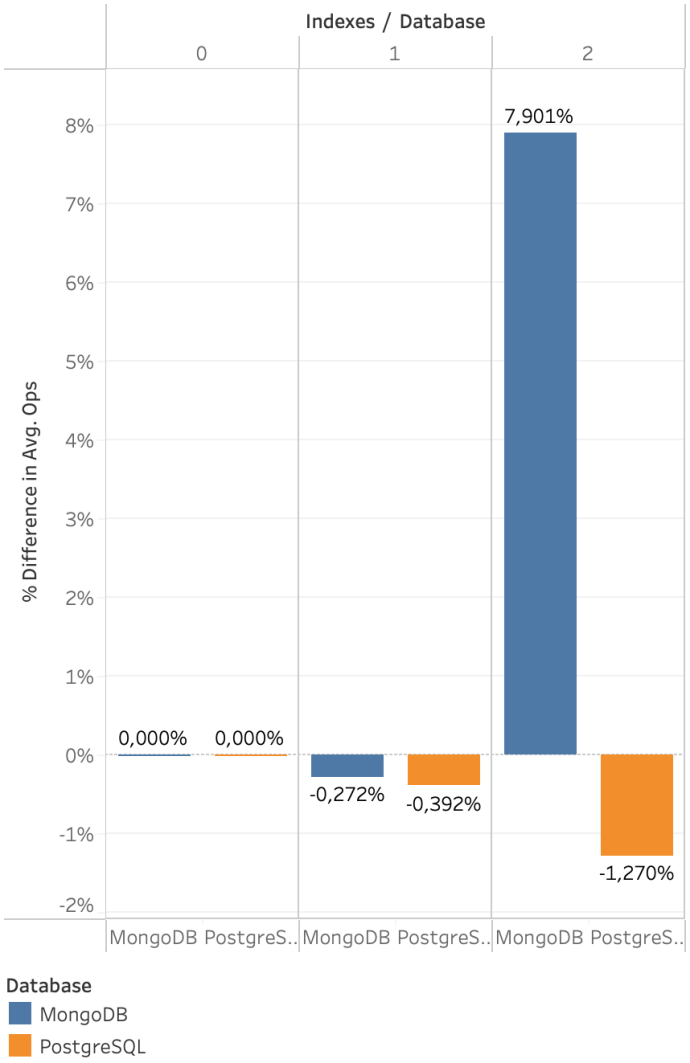


Figure 4.31: Impact of indexes for manual self-join

both exhibiting linear trends.

For fields queried, on the other hand, the region between two and four fields has a slight advantage for PostgreSQL. This advantage can be attributed to PostgreSQL's higher read performance, as this region is ideally in the middle of the number of rows that should change and the rows to query. The behaviour is nearly exponential for MongoDB, but PostgreSQL does not quite fit exponentially for the saturated region past four fields queried.

For write index experiments, all read indexes are present. Hence, the averages will appear higher, as seen in Figure 4.34. This is to measure the impact that only write indexes have. It can be seen that the number of fields changed is very similar to the read-index-only experiment. Fields queried has similar behaviour, though PostgreSQL seems to saturate earlier. This is to be expected since the introduction of write indexes requires more writes to disk and memory to update the data and the indexes.

Figure 4.35 shows the performance with no indexes. The results of the PostgreSQL experiment are what would be expected since the query part of the operation will be sig-

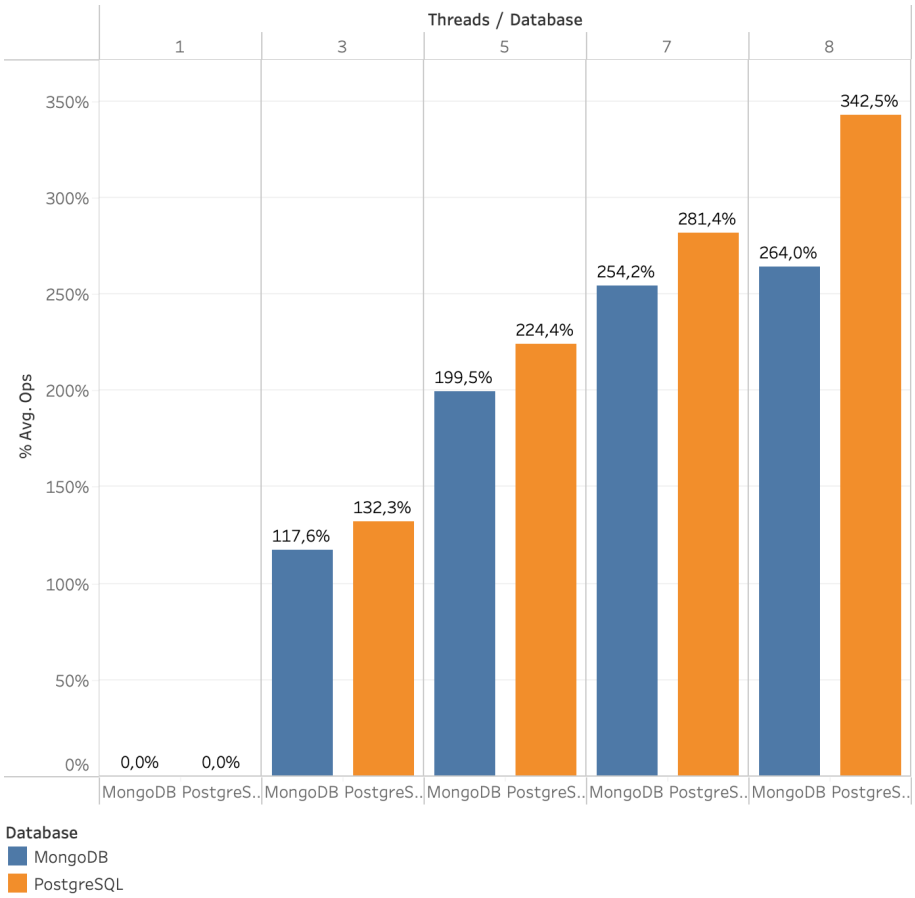


Figure 4.32: Impact of threads for joins

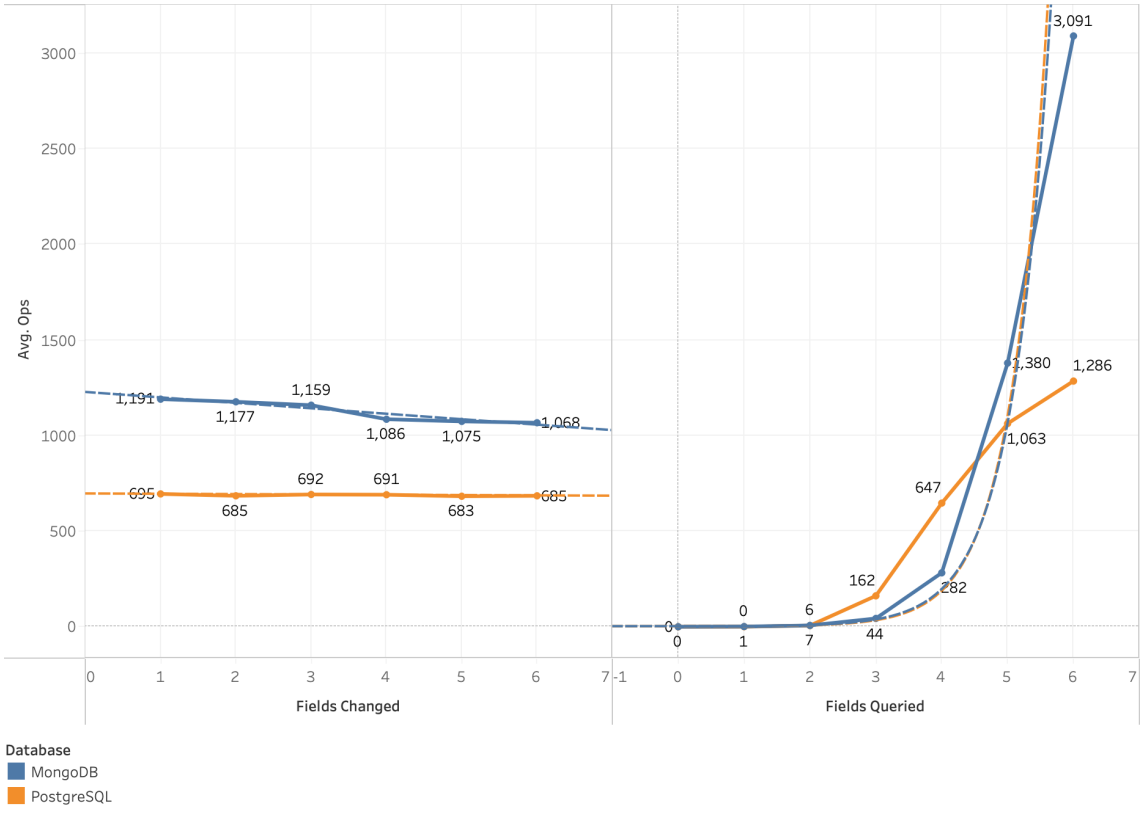


Figure 4.33: Average Ops/Second for updates with read indexes only

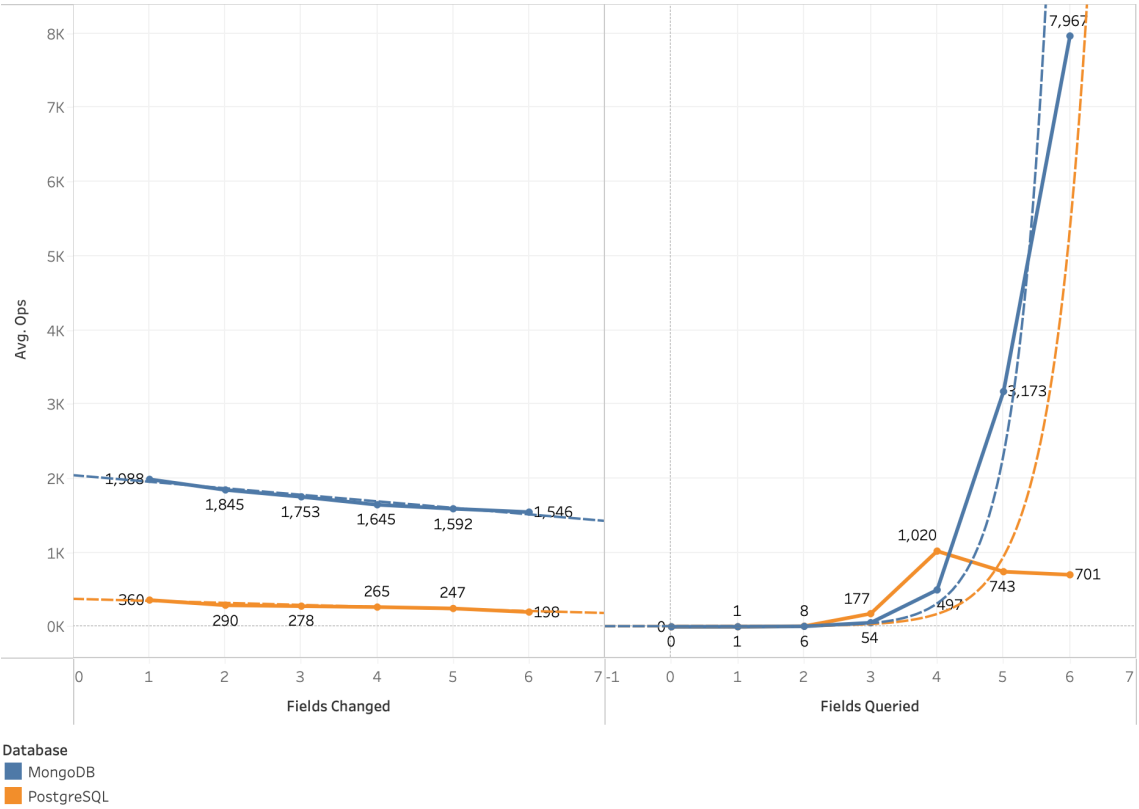


Figure 4.34: Average Ops/Second for updates with write indexes

nificantly slower. What is highly interesting is the fact that MongoDB performs better with no indexes, even without read indexes. This indicates that MongoDB always writes to all indexes, even if the field of the index hasn't changed. In this case, the read indexes too, which is why MongoDB performs better with no indexes. If there were even more documents in the database, a point would surely be reached where read indexes would be beneficial for updates.

Threads impact the update operations positively as expected. The impact is shown in Figure 4.36.

The impact of fields changed without write indexes is shown in Figure 4.37. In the case where write indexes are present, a more apparent decline in performance is seen as shown in Figure 4.38.

Querying more fields in these experiments limits the number of rows affected. Read indexes, in this case, help narrow down the correct rows more quickly. Furthermore, reducing the number of rows affected leads to fewer writes. Conversely, as seen previously, MongoDB still has to update all the indexes, even if not affected by the updates. Figures 4.39 and 4.40 show the impact of fields queried without and with write indexes, respectively.

The impact of read indexes only is shown in Figure 4.41. It can be seen that both databases do experience improved performance overall with the addition of read indexes. PostgreSQL is considerably so, and MongoDB significantly less. To illustrate the impact

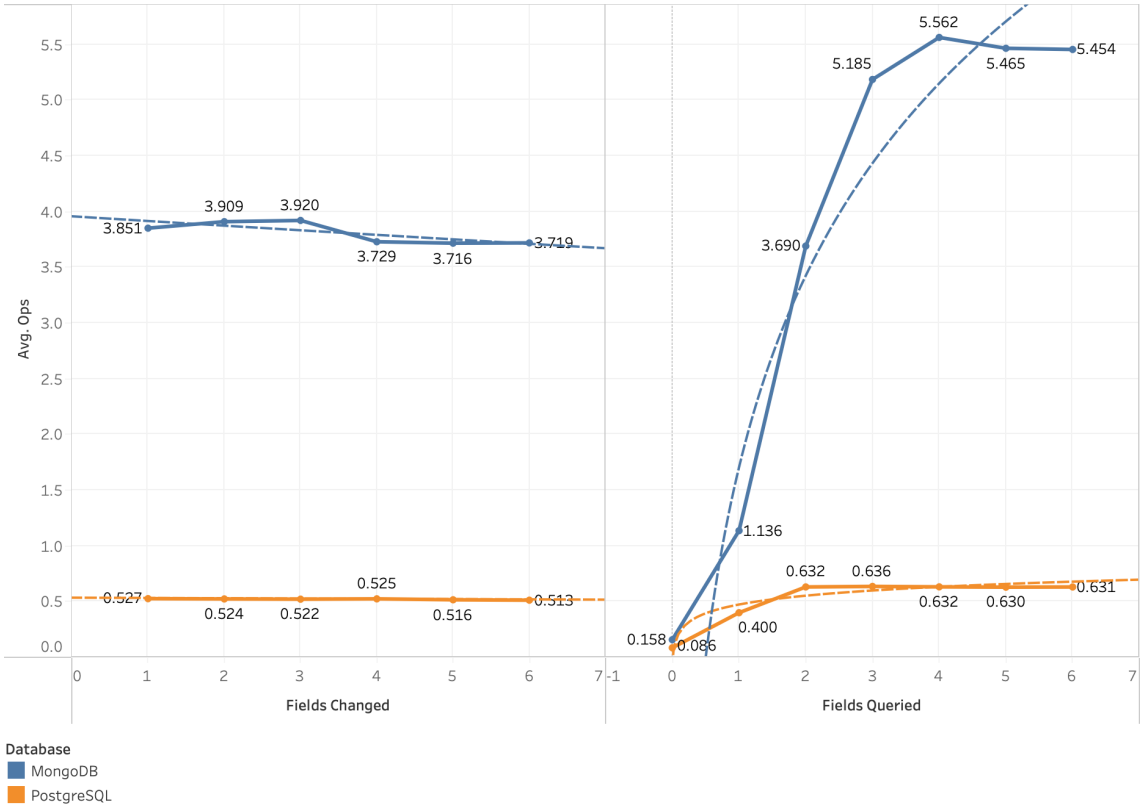


Figure 4.35: Average Ops/Second for updates with no indexes

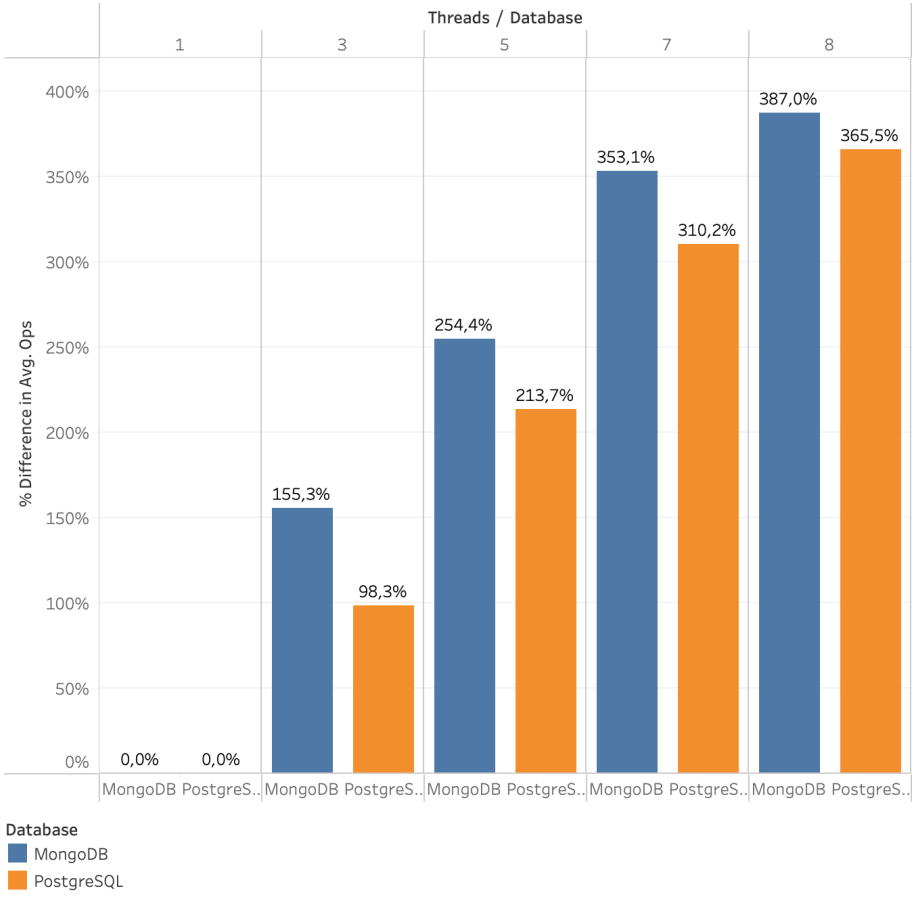


Figure 4.36: Impact of threads on update operations

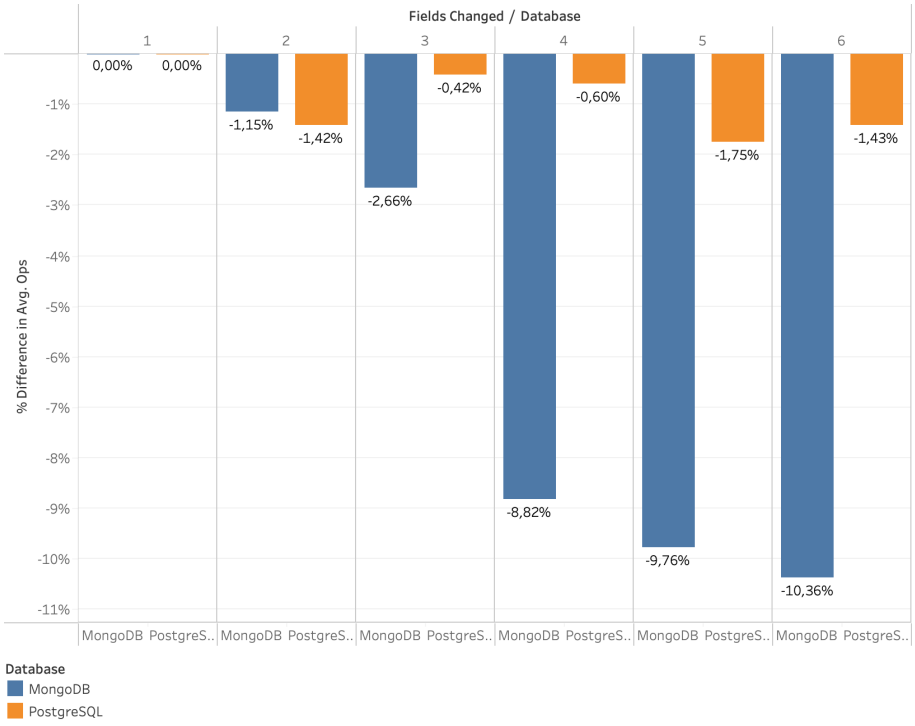


Figure 4.37: Impact of fields changed on updates without write indexes

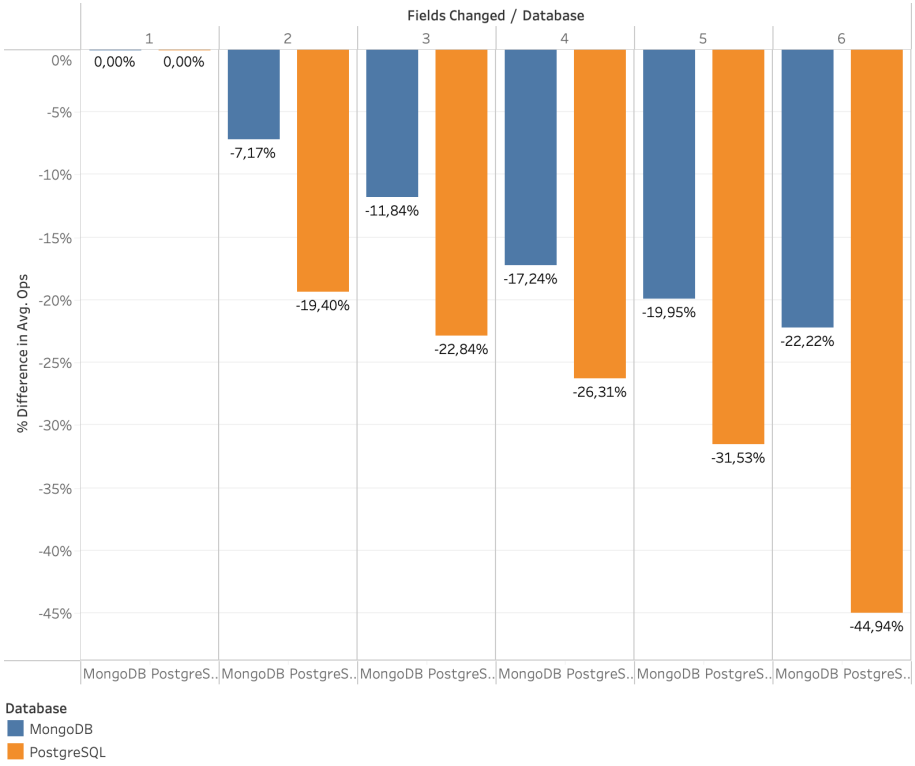


Figure 4.38: Impact of fields changed on updates with write indexes

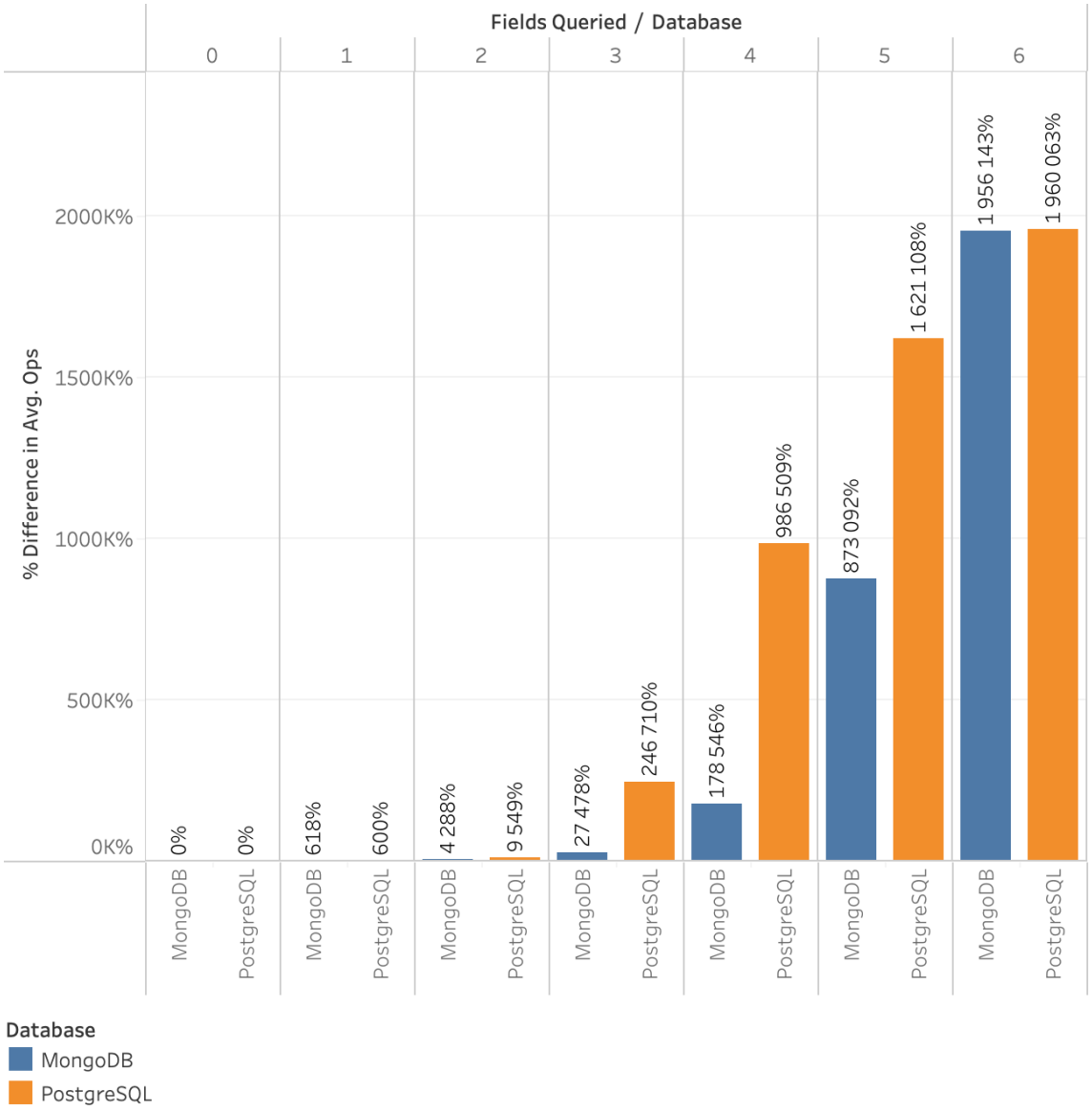


Figure 4.39: Impact of fields queried on updates without write indexes

of write indexes, Figure 4.42 shows only the impact of write indexes. In the latter case, both databases suffer equally from adding write-only indexes.

4.4.3 Summary

The experiments indicated that MongoDB performs better with updates on average than PostgreSQL. PostgreSQL does have a region - where the reads and writes are balanced optimally - that performs better due to its higher read speed.

The addition of read indexes improved performance overall, even though lesser so in the MongoDB case. It was shown that MongoDB updates all indexes even if unchanged, which describes the diminished effect of read indexes. Even so, the net impact of read indexes is beneficial.

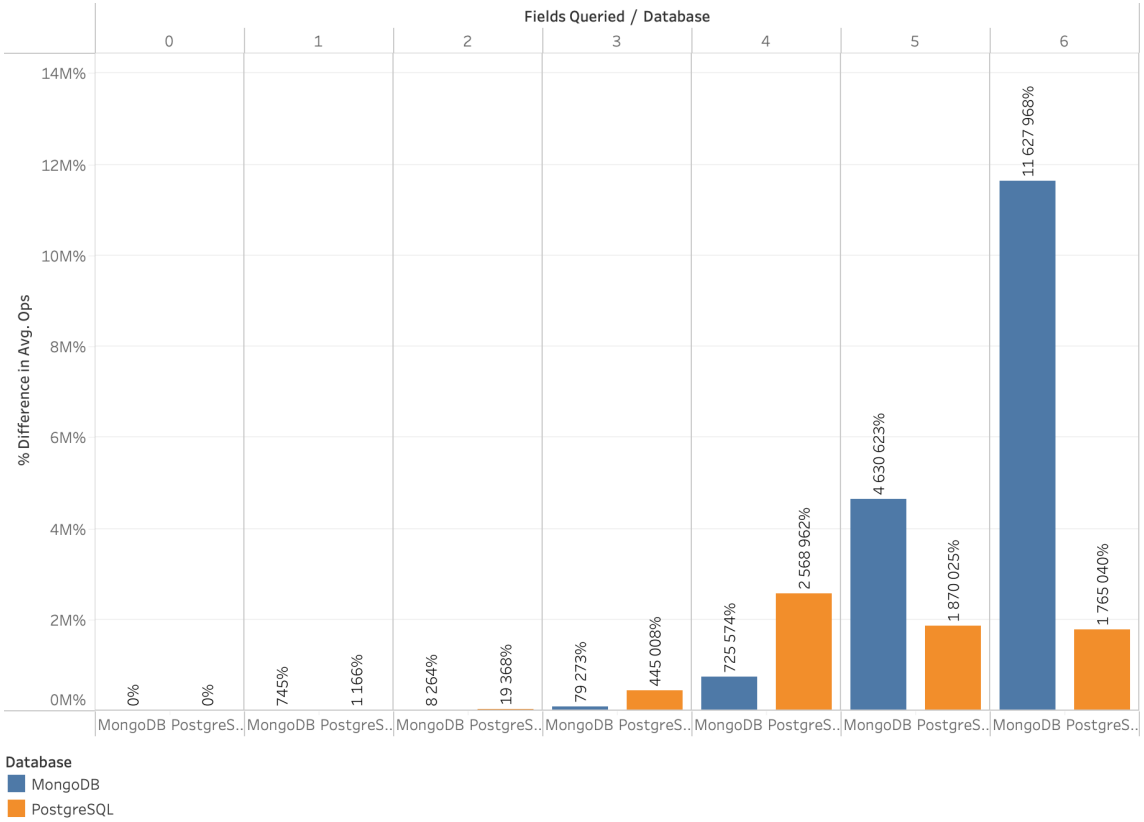


Figure 4.40: Impact of fields queried on updates with write indexes

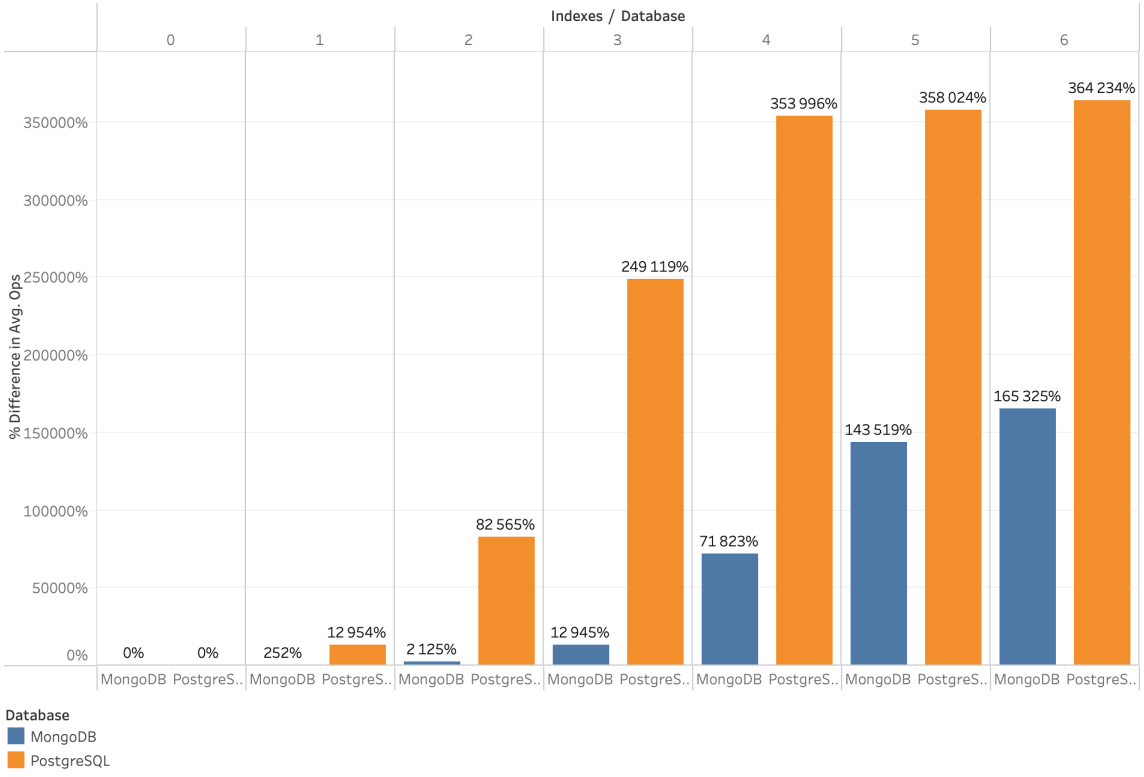


Figure 4.41: Impact of read indexes on updates

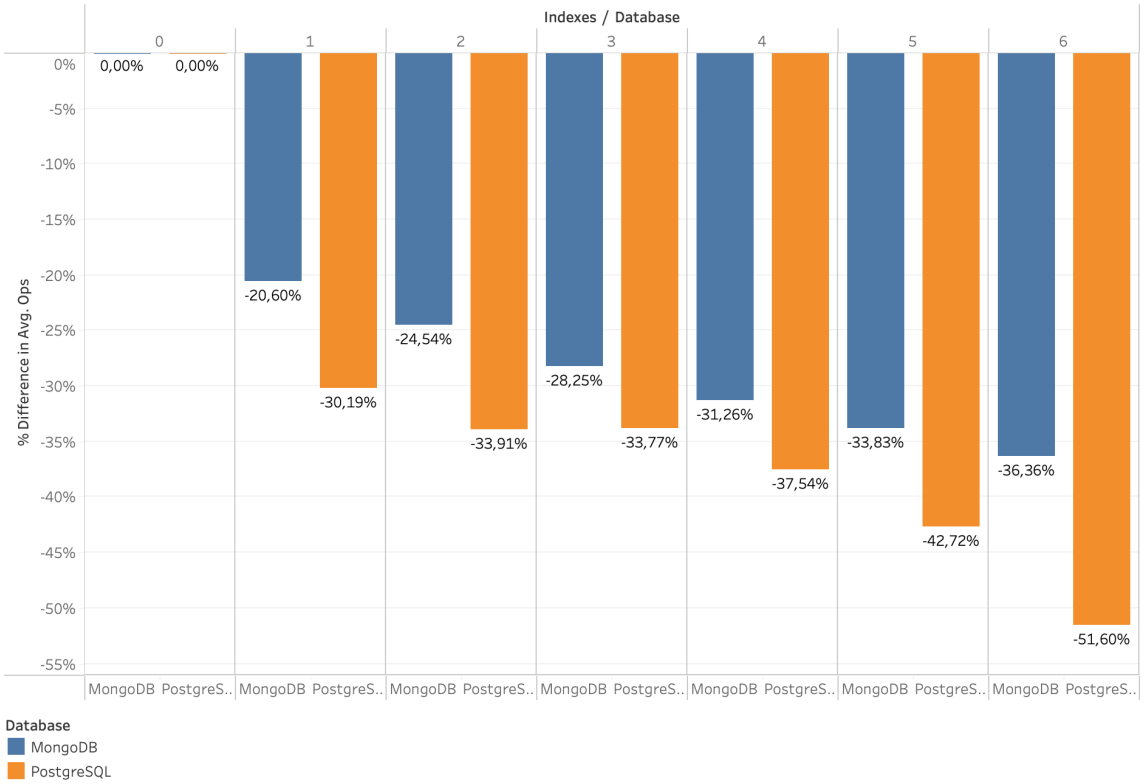


Figure 4.42: Impact of write indexes on updates

Write-only indexes, on the other hand, decrease the write performance across the board. The term write-only is in the context of a query, and it may very well be possible to have indexes for different queries interacting with all other queries.

4.5 Delete Experiments

4.5.1 Setup

Like update operations, delete operations contain read and write components. Rows or documents are wholly removed if matching the query, which has a waterfall effect on the indexes referencing the row or document.

The setup for these experiments requires that rows or documents be restored after each iteration, leading to a lot of idle time between test iterations.

Parameters affecting deletes are fields queried and a number of indexes. As with all tests, the number of threads was also varied. Table 4.9 shows the parameter values.

Table 4.9: Parameter setup for deletes

Parameter	Minimum Value	Maximum Value	Step/Increment
Fields	0	6	+1
Indexes	0	6	+1
Threads	1	8	+2

4.5.2 Results

As expected, the results closely resemble the behaviour seen in the update experiments. Figure 4.43 shows the results. Showing the effect of indexes alone, Figure 4.44 gives the results when only non-indexed iterations are averaged, and only the number of fields queried is changed.

The behaviour exhibited by MongoDB in the update experiments is repeated here for both databases since delete operations will always require all indexes to be updated. However, with no indexes to update, the gains are clearly visible. PostgreSQL's higher read performance shows to be prominent in these experiments.

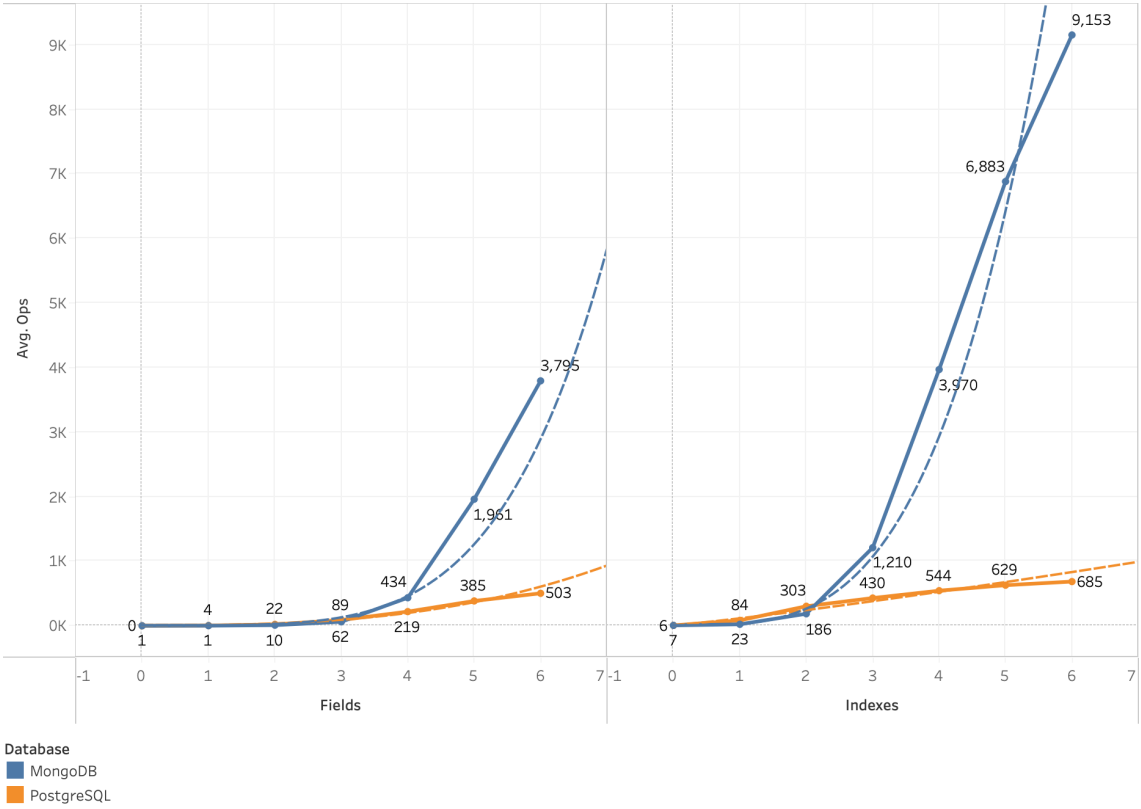


Figure 4.43: Average Ops/Sec for delete operations

To measure the full spectrum of the impact that fields queried have, the results for indexed and unindexed experiments are shown in Figures 4.45 and 4.46, respectively. As with updates, reducing the number of rows affected greatly increases the number of operations per second, much more so for MongoDB than for PostgreSQL.

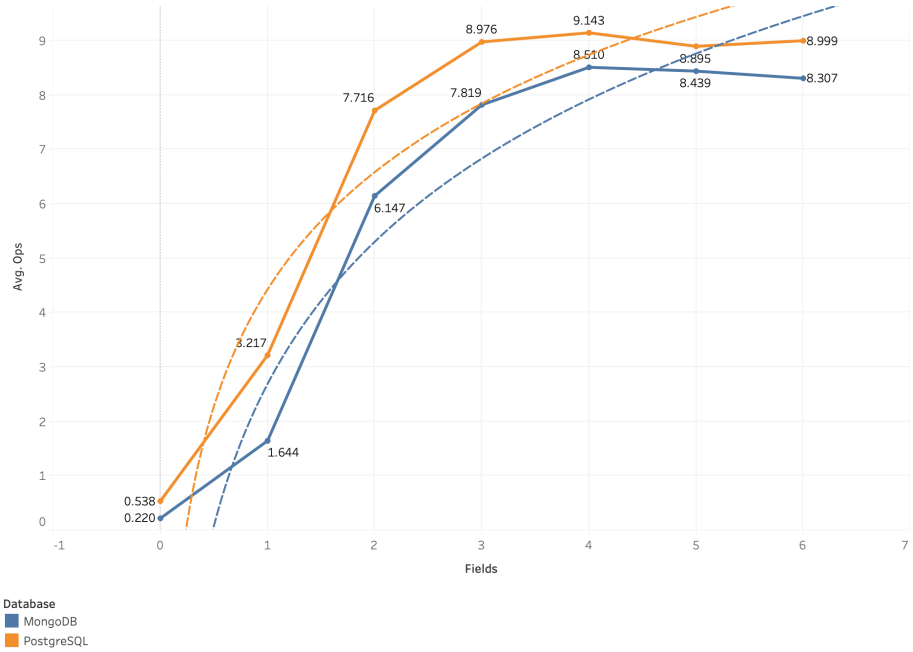


Figure 4.44: Average Ops/Sec for delete operations without indexes

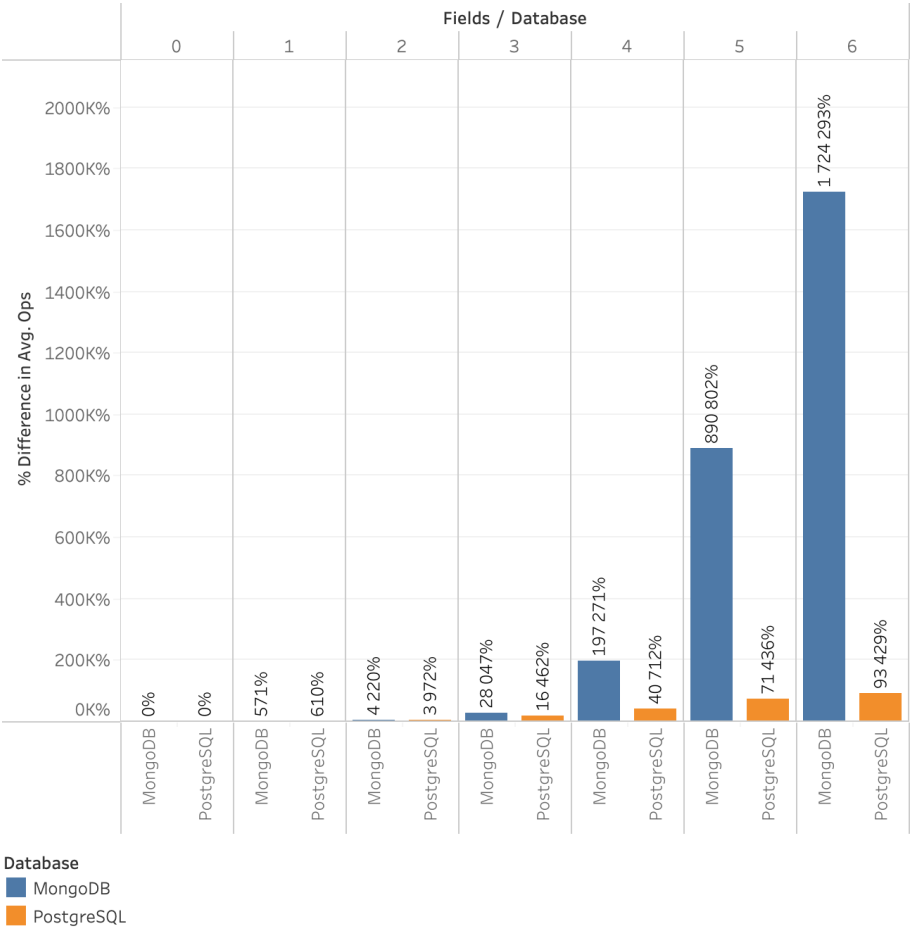


Figure 4.45: Impact of fields queried with deletes

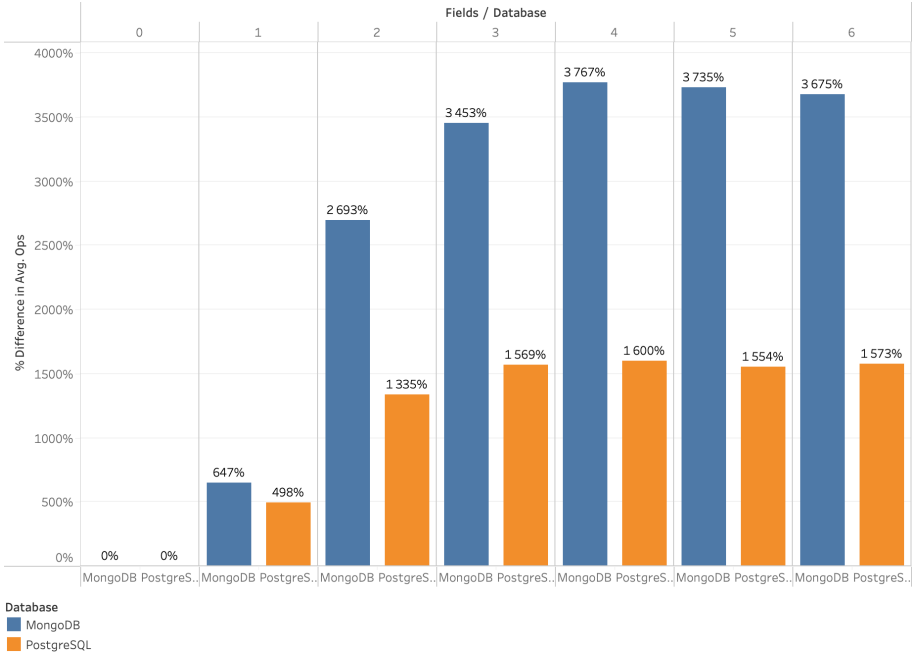


Figure 4.46: Impact of fields queried with deletes (no indexes)

Figure 4.47 shows the impact of indexes on delete operations. For both databases, it increases the performance. It must be noted that the fields queried were fixed on six to obtain these results. This was done to reduce the number of experiments required since the number of indexes should depend on the number of fields queried.

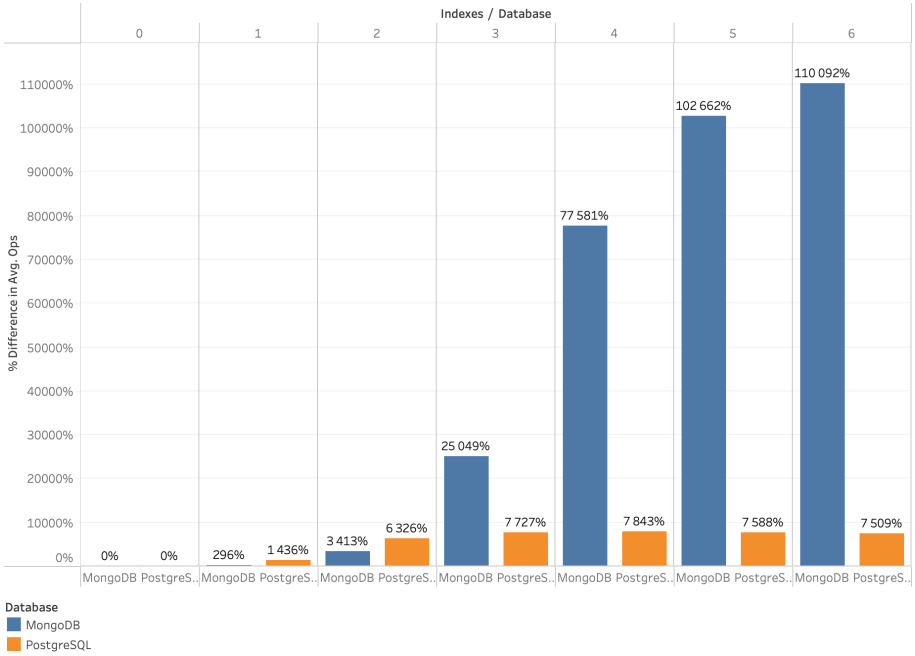


Figure 4.47: Impact of indexes on delete operations

Finally, increasing threads prove to, once again, improve performance significantly, as seen in Figure 4.48.

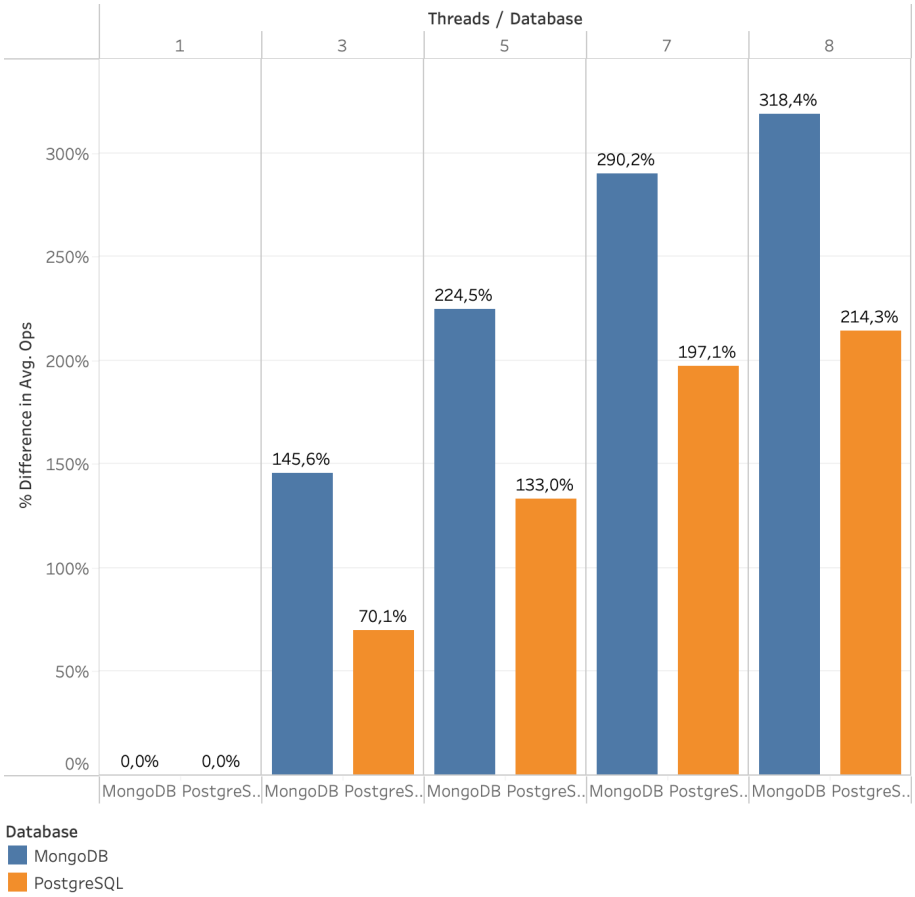


Figure 4.48: Impact of threads on delete operations

4.5.3 Summary

From the results, it can be summarized that both databases exhibit similar behaviour to varying parameters. On average, MongoDB performs more delete operations than PostgreSQL, especially with better selectivity (more indexes and fewer rows operated on).

Indexes, when used for querying, do improve performance for both databases. These experiments did not test the addition of write-only indexes, but the same behaviour seen in update operations can be expected.

4.6 Summary of Results

Across all experiments, the two databases exhibited closely related behaviour with some explicable anomalies. MongoDB generally performed better with inserts, updates and deletes - which are write-based operations.

In contrast, PostgreSQL performed better with reading operations, including subsets of write experiments which benefited the most from reading optimization. Regardless, the trends were very similar. It can be stated that the parameters influence the number of

operations per second predictably and of the same order of magnitude and form.

From the results, the following observations were made:

1. Indexes greatly improve read performance;
2. Indexes greatly improve sorting performance;
3. Indexes reduce write performance:
   - 3.1 Adding more indexes always reduces write performance;
   - 3.2 If the index helps reduce database scanning, the net effect is positive[4];
4. Having a high limit reduces the number of operations per second, though inversely increases the number of rows/documents per second;
5. Batching is beneficial in general;
6. Using multiple threads/connections is always beneficial[5];
7. Inserting more fields per row/document decreases write performance;
8. Returning fewer fields improves read performance;
9. Updating fewer fields improves write performance;
10. Querying with fewer fields improves read performance, even if indexed;
11. Write operations are far more expensive than reading operations.

4.7 Conclusion

From the empirical data, it is evident that parameter changes affect the two database systems comparably. Therefore, it can be concluded that the same design choices should affect performances similarly. This leads to an interesting situation where selecting the underlying database management system (technology) is less of a concern than the design of the data model and data access patterns concerning database performance. This is an important finding that was not foreseen at the onset of this research. However, it allows the designer more options and flexibility.

An important deduction can now be made, namely that the term “hybrid” could now imply a mix of traditional and modern NoSQL (denormalized) modelling techniques, regardless of the underlying technologies. This emergent knowledge is highly informative since it corresponds with the literature finding that the preliminary design phase has the

[4] For the specific index under question

[5] As long as the physical saturation point has not been reached

highest impact on the database performance. This supports the notion that the technology selection is done at the end of the SE preliminary design phase; that is, form follows function.

Important findings from the experiments include the effects of parameters on database performance. Knowing which parameters affect database performance, and more crucially by what magnitudes (relatively), and understanding that the process should consider application query patterns, combined with well-documented design methodologies, a decision support system can be synthesised.

From a research perspective, this chapter addressed the concept research solutions by providing decision-support information for the DSS process model. Importantly, the parameters that affect database design were identified to be used in the DSS decision tasks, which will be defined in the following chapter.

You don't get results by focusing on results. You get results by focusing on the actions that produce results.

Mike Hawkins

Chapter 5

Proposed Methodology

Introduction

The previous chapter characterised the DBMS performances for both relational and non-relational databases. These findings were used in this chapter to inform the development of the DSS process model.

It was found that both databases perform and behave similarly with parameter changes. Performance on essential database functions (empirical operations upon which all complex transactions are based) were characterised. It can then be safely assumed that changes in the design will affect both DBMSs in the same manner, considering their relative performance differences as seen from the experiments.

From the literature, it becomes evident that emphasis should be placed on the SE process's preliminary design phase. This chapter, therefore, focuses on this phase and applies the acquired knowledge from experimentation and academic literature. The relatively similar performance and dependence on design-dependent parameters will be used as decision-support information in the DSS design.

5.1 Overview

Not only database performance is essential when designing data-based systems, and systems engineering shows that considering each of the design phases individually is critical. This means the interfaces between the database, applications, and other peripheral systems must also be considered, including the associated resources, which will improve flexibility, maintainability and scalability. Moreover, factors such as human resources, technical expertise and existing infrastructure are also essential.

Traditionally database design is uncoupled from the application design due to the roles

of the developer and the modeller being separated [12]. Furthermore, this is further exasperated by data models commonly (often rigorously) adhering to the third normal form approach as the *de facto* standard. With the advent of NoSQL, where the data model more closely resembles code structures, and more specifically, the sheer growth and agility required by modern systems, this paradigm has been challenged [12, 20].

This chapter proposes a new methodology, which looks system-wide and designs for the full life cycle as advocated by SE. Once more, some empirical guidelines will be discussed for specific parameters but will be integrated into a system-wide paradigm which will help steer the design of data models in the early stages of the design process to a design which will be:

- Performant (by leveraging knowledge unearthed from the characterisation);
- Scalable (by incorporating best practices for distributed systems);
- Flexible (by making use of a modern paradigm for data modelling);
- Maintainable (by including the application and interfaces during design);
- Robust (by applying high availability controls and tools in modern database systems).

Database performance is quantitative and objective, and the characteristics were empirically determined in the previous chapter. Since more complex operations are combinations of empirical operations, a net effect can be predicted.

Emphasis is placed on performance engineering because it impacts a system's quality, reliability, maintainability and sustainability [31]. It can therefore be stated that there is a degree of interdependence between the design goals listed above.

Scalability can be tested by inspecting the complexity of relationships and operations to be performed, as discussed below.

Flexibility is more qualitative and subjective since both databases allow for changing the schema – scenarios that can guide a decision will be discussed.

Maintainability is another subjective factor, which can be determined by complexity, interdependence, required skill, existing architecture and technology, and integration requirements, among others.

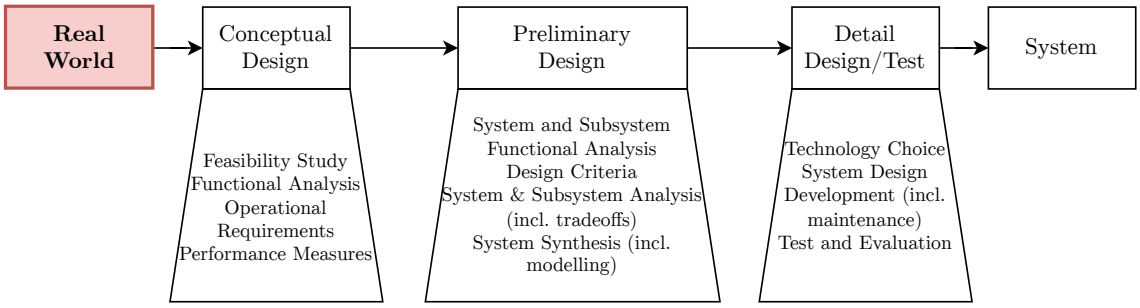
Robustness can be determined by examining the final design and determining risk factors and mitigations relating to high availability, which are also qualitative.

Since database (and application) performance is the only quantitative and objective measure[1], other factors will have to be determined by the weight of importance and trade-offs

[1] As far as behaviour and order of magnitude are concerned

for database performance will have to be made qualitatively. Each section describes possible trade-offs that may be required and what the expected impacts would be.

5.2 Phase 1: Real-world problem/input

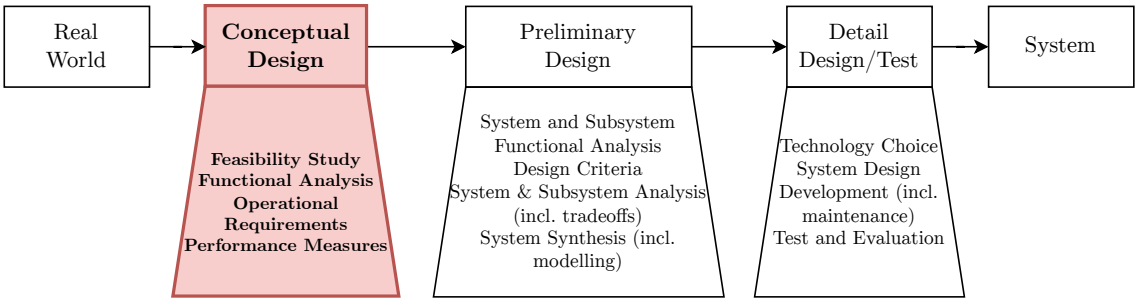


Considering the idiom that problems downstream come from problems upstream (also, garbage in equals garbage out), the requirements phase is critical and will significantly influence the stages to come.

During this phase, it is crucial to establish all the actors that will directly or indirectly manipulate and use the system (where data is concerned, at least). For each actor, all the possible interactions and actions must be elicited. All the high-level entities and attributes must be established from the data exchanges and activities [3, 41].

The above is obtained by performing a requirements elicitation at the system level [23]. It is vital to elicit requirements relating to the whole life cycle of the system as-is, to-be, and might-be. Otherwise, future growth could become inhibited and problematic.

5.3 Phase 2: Conceptual Design



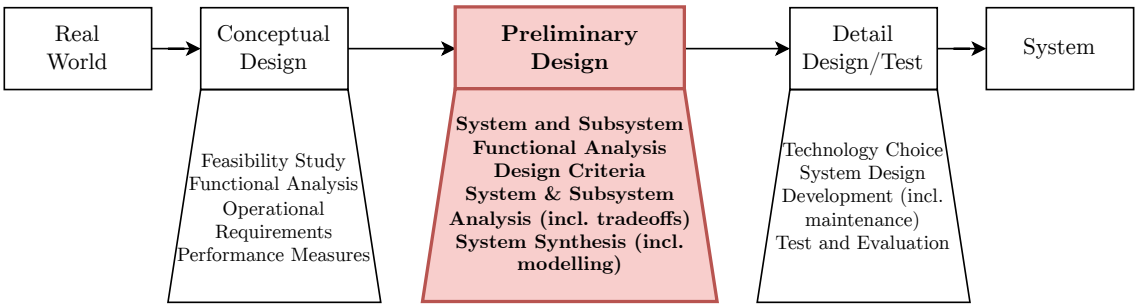
After the requirements have been thoroughly elicited, a high-level/operational-level functional analysis (SE) has to be performed. All the domains, scopes and technical performance measures (TPM) must be established. Statements regarding business rules such

as *entity A may only associate with ten entity B's* will prove helpful in making informed design decisions later on.

Carefully define each entity that may pertain to data-related design in terms of cardinality concerning other entities in a conceptual way.

Operational requirements must be established and will greatly determine the impact of specific functional units [23]. It is essential to point out that no database technology selection (technical requirements) or modelling has been made up to this point.

5.4 Phase 3: Preliminary Design



The most challenging yet essential decisions must be made during the preliminary design phase. Conceptual designs (ER) still closely represent the real world. On the other hand, preliminary designs more closely represent the abstract database system, while detail designs more closely represent the physical database system and, ultimately, the application's structures. In other words, the conceptual design closely resembles the use cases and actors, while the preliminary design resembles the application and system design.

The preliminary design phase mostly coincides with the typical tasks associated with the ER approach: conceptual and logical modelling. However, it is still to be done agnostic of the database technology. This section is split into two subsections: (i) Preparation and (ii) Refinement.

In the preliminary design phase, there are specific tasks and methods that will require the services of a skilled/trained DBMS designer as a subject matter expert. In addition, it is assumed that on projects of this scale, a systems engineer will form part of the development team in their customary multi-disciplinary capacity.

Preparation

SE methodology requires a refined functional analysis carried out at the subsystem level during the preliminary design phase [23]. The section below will focus on the database subsystem design, but imbuing the design with requirements and constraints that may

arise from other subsystems (such as the application) or functions that interface with the database is essential.

Design a use-case diagram as seen in the example in Figure 5.1. This diagram must indicate all the actors using the system and all possible use cases, including use cases that may become part of the system later. This diagram will be greatly enhanced by adding the subsystem functional analysis performed in the SE approach.

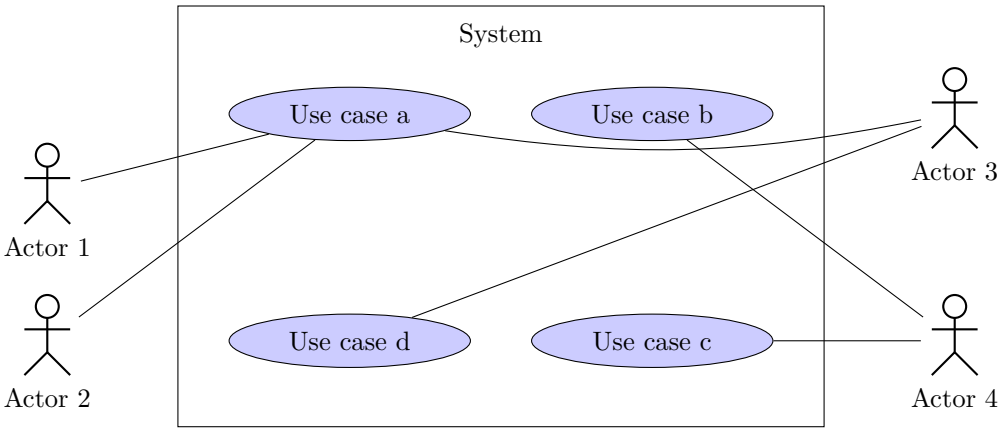


Figure 5.1: UML use case example diagram [118]

Include annotations pertaining to the frequency by which actors will invoke a use case (such as *once daily*, for example) or draft a UML activity (Figure 5.2) and sequence diagram that further documents the system [118].

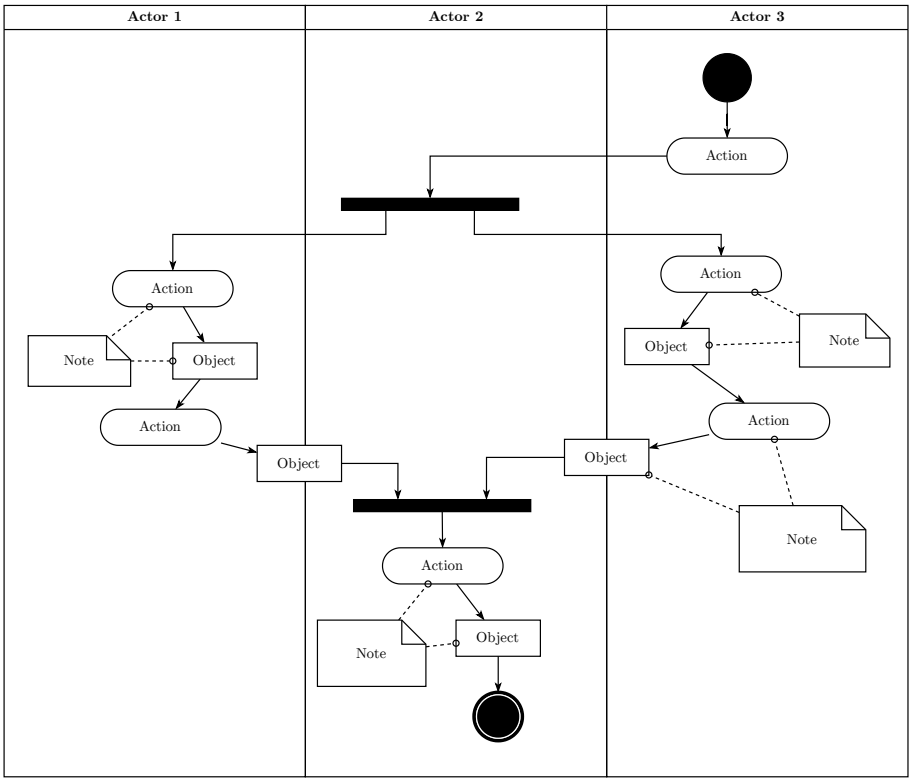


Figure 5.2: UML activity (swimlane) example diagram [118]

It is imperative to define each use case in sufficient detail to avoid missing critical relationships, attributes or requirements that could impact the detailed data design phase.

It is vital to capture all the high-level data-related entities, attributes and relationships identified from the requirements and functional analysis into a UML class diagram. Furthermore, the specific cardinalities extracted above can be annotated to aid decisions during later design steps. As mentioned before, cardinalities can be vague such as zero-to-many, one-to-one, many-to-many, or – more desirably – specific such as one-to-fifteen, for example.

It is of the utmost importance to make the distinction between the following kinds of relationships, from weakest to strongest, in terms of dependence [118, 162]:

Association
: Any logical relationship or connection in a broad sense without dependence on its existence.

Aggregation
: The parts form a whole, but both entities can exist without the other ex. a library stores books. The life cycle impacts have to be considered here as well. If the library was to be destroyed, the books might still exist independently.

Composition
: Composition is a stronger relationship where the parts cannot exist independently, ex. a building contains rooms. When a building is destroyed, the rooms are destroyed *by association* as well. Hence, the rooms' existence is dependent on the life cycle of the building.

Inheritance
: A specialization class inherits its parent's attributes, functions and relationships while gaining more specific traits. For example, an employee has an employee number and department, but a manager additionally manages employees (a different use case to model).

Association relationships should be carefully investigated and analyzed to determine if they might fall into one of the more specific relationships. However, relationships that do not are possible, for example, a supplier and client. Both entities have their own life cycle and do not fit the part-whole pattern[2].

An entity is then defined as a *strong* entity when its existence is not dependent on its relationship to other entities.

Also include annotations on the growth rate of entities, such as how many entities are added/removed per second, minute, hour or day. Query and update rates are also essential

[2] A library consists of books, but both can exist separately

and can be derived from related functions (or actions) and interfaces (the application, for instance) and their specifications [21].

Multi-tenancy is the ability to service multiple clients or accounts from the same database system. It is also important to annotate entities or relationships that discriminate the data for each tenant. This might be, for example, by the user and/or by account. For big data, it may often be required to add deeper hierarchies of the tenancy [163, 164].

Given the preliminary UML class diagram, activity diagrams, use cases and requirements, it is possible to synthesize a preliminary design. This must be an iterative process to address all the requirements described below. In essence, the process is now at the same point as a traditional conceptual model.

Refinement

Up to this point, it had not been necessary to consider the lower-level database format (tables or BSON documents). Again, it will not be needed during the rest of the preliminary design. This starkly contrasts typical logical modelling (middle phase of the design process), where ER/UML diagrams are translated to tables. There are some special considerations regarding normalization/denormalization, which will be discussed later.

The high-level entities are decomposed and fit into a logical data model for the preliminary design step. Logical data models closely represent the lower-level database form, whether document-oriented or tables, which can still be described in UML. Furthermore, relationships with no direct analogy in databases – such as many-to-many relationships – will be reduced in one of the many possible ways.

As demonstrated earlier in the thesis, rendering UML to Document-oriented or table-based databases is possible. Therefore preliminary design still makes use of UML class diagrams primarily. Simple elements with complex relationships are decomposed based on the ideal normalization paradigm given a full life cycle view *regardless of the underlying DBMS*.

Furthermore, each entity's attributes are garnered, and the attribute data type is specified in generic database types (strings, numbers, dates etc.). Here again, no physical database choices are made. For example, *string* could describe a description field instead of a physical DBMS qualifier such as `VARCHAR`.

Factors that may influence the logical data design choices in this process include the following:

- Number of rows/documents [101];
- Growth rate[3] [21];

[3] In terms of orders of magnitude, ex. one per day, 100 per minute or 1000 per second

- The lifetime of a row [6];
- Number and kind of relationships [6, 54];
  - Dependencies;
  - Dependents;
  - Type of association;
- Growth and spread of associations[4] [54];
- The cardinality of each relationship [163];
- Frequency and nature of access [11, 21];
- Frequency of updates [6];
- Frequency of amendment to the schema [45];

The following sections describe the above processes and discuss how these factors affect and guide the design during logical modelling.

5.4.1 Attributes

All entities' attributes (or fields) garnered from previous steps are to be annotated in the UML class diagrams. Basic data types can be used to annotate the fields initially.

Simple type attributes and some special fields, such as arrays of simple values, are to remain within the UML class and should not be further decomposed, especially if it is not considered a relational association (addresses or phone numbers of a user, for example).

All entities should be imbued with a unique identifier or ID field. The type for the field is not vital until the detail design phase, though typically, the ID must be unique to some extent, especially with regard to multi-tenancy.

5.4.2 Entities

Firstly all data entities and their attributes have already been rendered into a UML class diagram, along with their fundamental relationships and cardinalities, as established in the preparation sub-phase.

Considering the full system view, it is prudent that entities that will interface with the database represent *structures within the applications*. Again, this is typically not considered in the traditional database design paradigm. Therefore a refining process is required

[4] Consider a social media platform where a person can only add one friend at a time, for example

to render the relationships in such a way that they can easily translate to a lower-level database-specific format.

As a first step, entities that grow boundless should always be first-class citizens and not embedded. Boundless means that the entities can grow limitless in a relationship (or standalone), which implies they are difficult to embed fully. These entities will act as anchors for other entities based on the relationships and how they will be rendered. Suppose such an entity is a weak entity[5]. In that case, however, it could also be rendered with a bucketed pattern, as discussed later [165].

5.4.3 Relationships

Directionality

All relationships between two entities are, per definition, bi-directional. Though for querying and updating, it is essential to consider the direction in which operations are typically performed. The relationship can be considered single-ended if the data is always accessed in one direction only. This means that one would never, or rarely, query data in the other direction (starting from the other entity).

The relationship between users and addresses is an excellent example of a single-ended relationship. Though the address is technically related to the user, it would always be accessed via the user entity: one would not typically look up a user based on their address, for instance. To state it more plainly, it makes more sense to query the user to get their address rather than querying the address to get its user.

Directionality is often important to resolve cases where both entities of a relationship are equally strong.

Cardinality

Simplex relationships will first be discussed. Simplex relationships, for this methodology, are defined as relationships between entities that do not have more than one strong relation to other entities.

In the UML class diagram, full embedding (where an entity exists solely inside another entity) is modelled as a single field or array of fields of the class type that is being embedded. In contrast, referencing is modelled by only adding a *foreign key* field or array of *foreign key* fields. These must be of the type that the referenced class' *primary key* or ID field is.

[5]Existence is based mostly on relationships with other entities

One-to-one Relationships to entities that are of the one-to-one variety should be annotated with the associated class as type. Generally, one-to-one relationships are to be embedded, especially when there is a robust and natural affinity to one end. A good example would be an address of a user. It could be modelled with separate entities for addresses and referencing the address from a user entity, but it is more natural to embed the data. These relationships are often single-ended, as in the example where a user could still be queried directly via his/her address if the address field was indexed. One would still not require a reference in the opposite direction when embedded, even in cases where the address is duplicated.

One-to-many Making a distinction based on more specific cardinality is essential for one-to-many relationships. If the right-hand of the relationship is boundless, the right-hand entity should always reference or embed the left-hand entity. For bounded or few[6] right-hand elements per left-hand element, it might be beneficial to fully embed the right-hand elements in the left-hand element [166]. The rule-of-thumb here is that *composition or inheritance should typically be modelled with embedding* – the right-hand entity would embed the left-hand entity. In contrast, typical association or aggregation should reference the left-hand entity, especially in complex relational cases (where an entity is related to more than one other entity type).

Even when considering the possibility of multiple addresses per user in our example (which is technically one-to-many), the choice to embed the data seems *more natural and beneficial*[7] since atomicity is still maintained, and queries can be made from both ends, given appropriate indexing where applicable. Thus they are still *functionally* equivalent, but the data is related to the user, and only the user is closer to home. Furthermore, this more closely resembles how the data would be represented in a typical object-oriented programming language[8].

Many-to-many Many-to-many relationships come in three varieties, each with different approaches required [162]:

Many-to-many

This kind of relationship has too many entries to consider embedding on either end. For these relationships, it is best to have a traditional `JOIN` entity that pairs the left and right-hand entities. These should be referenced, and full embedding should not even be considered a baseline design choice.

Many-to-few and Few-to-many

Contrary to a many-to-many relationship, many-to-few and its counterpart have only a few entities to reference or embed on the other end. This can then be handled

[6] Less than a few hundred for practical purposes can be considered one-to-few as a rule of thumb
[7] No `JOIN` would be required. A single row lookup could satisfy most queries
[8] Very few languages use tables and rows

similarly to the one-to-few modelling described above. No `JOIN` entity is needed to fulfil this requirement. Typically these are rendered as an array of references rather than full embedding to reduce duplication and improve consistency. However, a balance might be found with the *extended reference* pattern described later on.

Few-to-few
Similar to many-to-few, few-to-few relationships can embed or reference on both ends, depending on the association type. One usually stores references on both ends to reduce redundancy and gain strong consistency.

Complex relationships

Only simple situations have been considered up to now. When more complex relationships exist where entities have different cardinalities and association types with more than one entity, one should carefully consider how to render each entity and relationship.

The above process should be followed recursively, rendering the stronger relationships first until only simplex relationships remain. Considering that boundless entities serve as anchors, start with these and generate each association accordingly.

The primary decision to be made when modelling entities is whether to embed or reference data or employ a hybrid approach. It is here where application query patterns play a vital role in determining the necessary trade-offs to consider.

When fully embedded in complex cases, the following questions have a bearing on the design choice:

- How often is the embedded data changed?
- How often is the embedded data required in a query?

The answer to these questions will determine whether embedding is ideal. When embedded data changes often, these changes must be fanned out to all other entities embedding this item (if consistency is required). If the ratio to writes versus reads is skewed to the write-heavy side, referencing might prove more performant and maintain a stricter level of consistency [166].

However, if the ratio is skewed towards the read-heavy side, embedding might prove more performant by removing the need to `JOIN` the entities during a query [166].

The characterisation showed that writes are more costly on a database than reads, and it should be given priority when designing these kinds of complex relationships.

Referencing is typically better for scaling writes for complex cases, primarily by avoiding the onset fan-out. Embedding, in contrast, is better for reading performance, as it avoids

`JOIN` operations. A third option is to utilize both models, following an eventual consistency approach to embedded data. Attributes typically required in queries should be noted as possible candidates for partial embedding for this approach. A fourth option could be to utilize the bucket pattern (described later on), which creates *buckets* of references.

5.4.4 DBMS Choice

From Chapter 4, it was found that the specific technology was not the primary determinant of the overall system performance. This was clear from the experimental results. The premise of this guide (DSS) is to perform the conceptual and preliminary design agnostic of the underlying technology. The technology selection can be based on the findings from the experimental results.

Therefore, a database system (or systems) technology must be chosen at the end of the preliminary design phase. This is done by performing a trade-off study to address design and operational effectiveness. The system's constraints, controls, resources, inputs and outputs should dictate the choice. Since form follows function, the database chosen should have a form and fit to meet the function [167]. This section will guide the choice of underlying DBMS(s).

The characterisation of the two DBMSs chosen for this thesis demonstrated both systems perform and behave similarly except for a small subset of parameter choices. In general, PostgreSQL performed better at reading operations, while MongoDB performed better with write operations, and at the end of the trade-off study, this will be considered. Moreover, from the literature, it was seen that both databases provide similar features.

Disregarding database performance and scalability, for now, some qualitative and subjective factors influencing the DBMS choice are summarized below. These, in the form of a value system, could be used for multi-criteria decision-making (MCDM) based on a weighted criteria approach [168]:

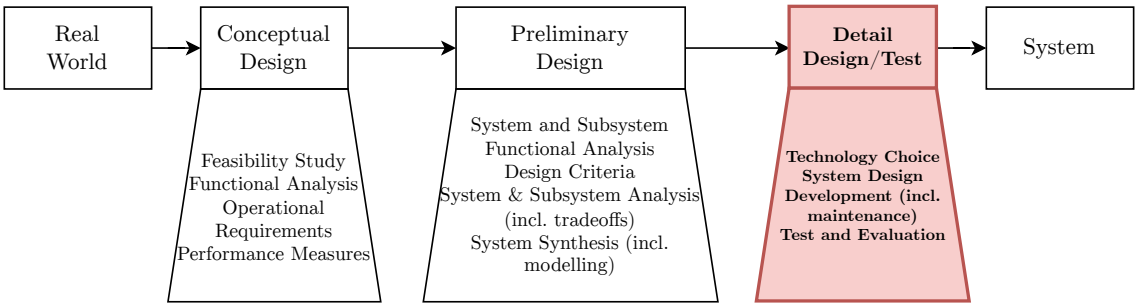
- Institutional knowledge in the form of a recorded body of knowledge (information);
- Skilled developers and designers on a specific technology (human resources);
- Available technology in an organisation (commercial or convenience);
- Infrastructure that support a specific technology (including agreements);
- Support from a specific vendor or service provider (specialised support);
- Legacy systems (AS IS) that may simplify transfer of data (portability);
- Future systems (TO BE) that may require a specific technology (need);
- Access to database maintenance / management personnel (administration);

- Additional factors outside the scope of this research.

The factors above primarily affect the choice of the underlying DBMS in a qualitative manner and will most likely be subjective. It is important to note that the data model also dramatically impacts the decision and should be kept in mind when evaluating the system. Details on the above factors, that affect the DBMS choice, fall outside of the scope of this research and are best determined by stakeholders that hold an interest in the project.

Since this guide serves as a methodology, it is possible to substitute any two DBMSs. However, characterising the database options and following the same paradigm would also be required when comparing different two. In this guide, MongoDB and PostgreSQL exhibit the ability to embed and denormalize data in a NoSQL approach.

5.5 Phase 4: Detail Design



After the preliminary design phase, a specific database system or systems have been chosen based on a trade-off study. Generic constructs such as entities must now be represented by tables (PostgreSQL), documents (MongoDB), or, in the hybrid case, both.

After the entities, relationships and attributes are rendered, some architectural decisions relating to availability and scalability must be made. Therefore two sections are presented, a development section and an architecture section.

Development

5.5.1 Key Fields

Candidate Keys

Keys, or fields and combinations uniquely identifying an entity, are called candidate keys. Any candidate key should uniquely identify any instance of an entity. An essential attribute of a candidate key is that it should not contain any other keys. The primary key

is chosen from the pool of candidate keys and should be unique for the system's lifetime. For example, a user might have a username and email address; however, the system does not allow a user to change their username, while the user may change their email address [2].

Primary & Foreign Keys

Each entity should have a unique identifier that is *primarily* used to identify it. Traditionally these fields have been called primary keys, used to reference the entity in its relationships.

In the user example from earlier, it was found that the *username* field was more suited for the primary key since it does not change during the system's lifetime. This is an essential factor to consider [3].

Primary keys do not have to be a singular field and may contain compound fields, or as shown by Copeland, a single compound string containing a prefix-searchable compound field [11], for example.

In MongoDB, each collection entity *must* have an `_id` field which is automatically indexed uniquely[9]. Furthermore, the field is immutable, and the record has to be deleted and recreated with a new `_id` field if needed. MongoDB will, by default, add an `ObjectId` value, which is date-sortable and globally unique [48].

Similarly, PostgreSQL and relational DBMSs typically have a `UNIQUE NON NULL` property added to the `PRIMARY KEY` attribute, which prevents duplication of primary keys. This is not strictly enforced, and it is possible to create tables with no primary key, though it is a recommended best practice to have a primary key per entity [51].

Primary keys stored as a reference in another entity are called foreign keys. Foreign keys could also be a component in a primary key, especially when it is part of a `JOIN` table [52].

5.5.2 Referential Integrity

Referential integrity is the part of an ACID system that guarantees references remain consistent. That implies that an entity cannot be removed or changed while still being referenced by another entity via foreign keys. Per relationship, constraints can be applied to foreign keys to dictate the operation to follow. The options for updating and deleting are summarized below [51, 52]:

`ON UPDATE/DELETE`:

`CASCADE` operations will apply the changes downward: if an entity is deleted while refer-

[9] Capped collections are an exception to this rule

enced, the entities having references will also be deleted/updated.

`RESTRICT` operations will prevent the changes and cause the transaction to fail.

`SET NULL` operations will set the references to `NULL`.

`SET DEFAULT` operations will set the references to the default value for the reference column. `NO ACTION` is similar to `RESTRICT`, but the action is deferred until later in the transaction.

`NO ACTION` does nothing.

Although the above are executed within the database system, specifically PostgreSQL, similar guarantees can be implemented and managed from the application with a simple update transaction per entity/relationship. Application side referential integrity can be applied to PostgreSQL and MongoDB, though it is the only method possible for MongoDB.

5.5.3 Indexes

From the characterisation done in the previous chapter, the factor which had the most significant impact, bar none, is the choice and application of indexing. It impacts both write and read operations and should therefore be carefully and meticulously designed to support the application's specific needs and requirements *for its full life cycle.*

Indexes have only one primary purpose: to speed up reading and sorting. These include read-related aspects associated with updates, deletes and inserts under constraints[10]. Benefits differ by magnitude based on the number of rows affected by an operation. The more rows/documents that have to be found and sorted, the greater an index's impact will manifest.

It is, therefore, important to know how the number of rows/documents grows in the full life cycle of the system. If a predictably small number of rows will always be present, indexing might not be crucial.

Conversely, with each additional index, write performance suffers for all write operations to a table or collection. This is true even when the fields that are updated are not indexed[11]. However, the orders of magnitude in losses are significantly smaller than the gains that reads inherit.

To successfully design the indexes, one has to consider the frequency at which queries will be performed. It might be more desirable to allow longer query times than to suffer lower write throughput for all inserts/updates. However, if a query is performed frequently,

[10] For example, inserts that have to ensure uniqueness on one or more fields

[11] This mainly occurs when a document has to move physically on the storage system – all indexes have to be updated to reflect the new physical storage address

especially if the collection/table grows significantly large, it cannot be helped but to add an index [11].

When it is conclusive that an index *is* required, it is then imperative to properly design the index/indexes.

Binary Tree Indexes

Binary tree indexing is the *de facto* standard indexing technique for MongoDB, PostgreSQL and most common DBMSs. B-trees, as they are referred to, store key-value pairs. The key would be the indexed column or field, and the value would be where that data element physically resides [54].

B-trees can be looked up based on an exact key match, a contiguous range of keys, or a prefixed key [54].

These indexes can also be compounded with multiple columns or fields, which have to be queried in order – that means if the index is built with column A, then B, then C, one cannot look up a C value only. Queries must be A first, then B, then C, or ranges thereof, to benefit from the index. Furthermore, postfixed queries cannot be used either, and columns cannot be skipped [54].

This character of B-trees leads to the rule of thumb of indexing multiple columns – *start with the most selective column first, then work toward the least selective column.* This must be superseded by the fact that the left-most columns have to be fully specified to benefit from the right-most columns [54].

It is important to note that individual indexes on columns cannot be used together on multi-column queries. Therefore, it is crucial to structure the index appropriately and get as much mileage out of a single index as possible. It might also be prudent to consider querying with additional fields to utilize existing indexes [54].

Another factor to consider is that if results have to be ordered, the index ordering should coincide with the query order requested [54].

Speciality Indexes

While MongoDB and PostgreSQL can index similarly, PostgreSQL has the most index types, with various features and caveats. However, for this thesis, only common indexes will be discussed.

Partial Indexes Both database systems can create partial indexes. Instead of indexing over the entire key space, these indexes only index based on specific constraints. An

example would be the unique constraint or indexes, which only index rows/documents that contain a particular range of values (for instance, only accounts in arrears).

The advantages are that indexes are smaller and queries that hit the index have a much smaller scan surface to search through than full indexes. It has no specific negative impact, except that it cannot be used for sharding or partitioning. Furthermore, they can only be used on queries that satisfy the same constraints, which limits their general usefulness [48, 51].

Geospatial Indexes Many modern applications are location-aware, from advertising to IoT and all in-between. One needs a geospatial index to efficiently query based on proximity, containment, and intersection on spherical or two-dimensional coordinates and shapes. Both PostgreSQL and MongoDB support these indexes and queries [51, 76].

Full-text Search Indexes Like geospatial indexes, if an application is required to do a full-text search, one needs an index to accommodate the queries for a growing database. The critical difference between standard and full-text searches is that it is language-specific and supports efficient querying of words within more extensive texts. These indexes support diacritic and case insensitivity and some limited support for collation [48, 51].

Dedicated solutions can work well for hybrid integration and are purpose-built for full-text search. It is often a more desirable solution if a full-text search is a critical application function. Popular open-source solutions include Apache Solr[12] and ElasticSearch[13] [11].

Covered Indexes Covered indexes are not a special kind of index but specific conditions where additional fields in an index improve query performance. When a query and projection can be fully satisfied with the index alone, the database system does not need to fetch the actual row/document. It can completely fulfil a query by itself [48, 51]. It is, therefore, a possible optimization to include an additional field or fields to the index to satisfy the covered condition. Adding more fields impacts write performance, as demonstrated by the characterisation.

5.5.4 Design Patterns

This section might have an impact on the preliminary design phase. The former sections have discussed typical modelling techniques. Specific patterns exist, however, which should be implemented where needed to make optimizations [165]. These patterns impact design choices for both the database and the application, reinforcing the system-wide design concept.

[12] `https://solr.apache.org/`
[13] `https://www.elastic.co/elastic-stack/`

Approximation

Approximation is a pattern that periodically, or when a certain threshold is met, updates data from the application side as needed. This pattern reduces high velocity and turbulent updates to the database and is helpful for data that is not mission-critical for to-the-second accuracy. A good example that could benefit from this kind of pattern is counters of most types, where knowing the exact number is less important than knowing a ballpark (or eventually consistent) approximation.

From the experimental characterisation, it was demonstrated that batching is always beneficial. This pattern can exploit this behaviour, for example, by scheduling periodic batch updates of statistics and counters instead of using many individual updates.

Attribute

In the case where many fields are of the same context and have to be searchable with single queries, this pattern is helpful. Indexing many fields is expensive and, as shown earlier, challenging to make into a compound index. Therefore, instead of creating many indexes, the attribute pattern suggests grouping the fields within a single array indexed on a shared internal field.

This pattern can be implemented on MongoDB and PostgreSQL, though depending on the complexity, PostgreSQL might need to use `jsonb` columns.

Bucket

Bucketing is a pattern where one row or document stores an array of elements instead of fanning out on rows or documents. A new *bucket* is added when the array reaches a specific size. This pattern allows for higher-arity embedded relationships; by linking to a parent node, one can page through many buckets.

Computed

When complex operations are required, but real-time accuracy is not critical, one can leverage the computed pattern. When the associated fields are not frequently updated, one can improve read performance by periodically pre-computing a complex field. The level of accuracy can be determined by, for example, per-minute updates and daily recalculations.

Versioning

The versioning pattern uses two collections or tables to store the current version of a document/row and revisions of the same document. This pattern is applicable when there are typically few revisions per document, few documents and most queries will be on the current version.

Extended Reference

Typically a reference will only contain the foreign key of the referenced element. The extended pattern embeds frequently joined fields that do not change often or where eventual consistency is acceptable.

Outlier

The outlier pattern can be used in cases where a relationship is typically one-to-few and very seldom one-to-many. The pattern uses embedding for the general case, referencing exceptional or outlier circumstances. Since MongoDB does not enforce a strict schema, it can easily be malleable in these cases. For PostgreSQL, one can use a *nullable* field for both the embedded entry and the referenced entry or a flexible column like the `jsonb` type.

Polymorphic

Polymorphic collections or tables allow for documents or rows with many fields in common to reside in the same space. For schema flexibility, PostgreSQL can use `jsonb` fields or inheritance if more rigidity is acceptable.

Schema Versioning

This pattern only applies to MongoDB or PostgreSQL's `jsonb` types. It is prudent to add a schema version field in a document for flexible schemas that lose backward compatibility. As a result, the application can alter its processing accordingly, reducing application complexity.

Subset

The subset pattern is very similar to the extended reference pattern. A document embeds a subset of the relation and references the rest. This is useful, for example, to keep a cache of sorts, for instance, keeping a user's last ten logins. The information is immediately

available without `JOIN`ing, and if a drill-down is required, the application can leverage full reference-based queries.

Tree and Graph

One can embed an entity's ancestry in an ordered array of hierarchical data that must be stored. One can also store direct parents/children separately for convenience. This pattern lends itself to product categories very well.

5.5.5 Views

PostgreSQL and MongoDB support views, which are read-only query-based lookup functions, presented as an actual table or collection. Views are only pre-established queries and projections on one or more tables or collections. Views have the advantage of taking a form already usable by the application without having to duplicate query code. In the case of MongoDB, views are built from an aggregation pipeline, while PostgreSQL makes use of standard SQL queries [48, 51].

When a query is run on a view, it can be materialized into a physical table or collection. A materialized view is often used to generate and aggregate periodic information. Materialized views can be indexed to improve query performance and updated by merging in new results. This is similar to the computed pattern mentioned above but works on table or collection level typically [48, 51].

5.5.6 Consistency

Denormalization has an impact on the consistency of both database technologies in the following ways:

- Updates have to fan out to prevent stale data;
- Data is always eventually consistent (given updates are performed on multiple rows/documents);
- When manual sharding is applied.

Strong consistency can be achieved using transactions on MongoDB and PostgreSQL but should be used sparingly and only where necessary.

The trade-off is to consider whether a field of an embedded entity is updated often or is usually required in the results. Good candidates are seldom updated, but frequently requested fields [166].

There is also the ability to make more robust consistency queries when having current information is critical. In these cases, the original document or row can always be queried, instead of relying on the embedded, possibly stale data.

One can rely on transactions for multi-row or cross-collection/table updates to provide strong consistency. One can also use two-phase commit protocols, which guarantee that if a portion of a transaction fails, all changes made can be rolled back [41]. Many databases make internal use of this protocol to accomplish transactions, but it can be implemented on the application side as well [6].

MongoDB has guaranteed atomic operations per row, which, when embedding, can provide atomic operations across more than one entity. This is a special case where MongoDB can provide strong consistency with a modern paradigm.

It is clear that an appropriate consistency level can be achieved in different ways for PostgreSQL and MongoDB, and that it forms part of the modelling and design steps. During the detail design, a specific consistency approach should be chosen on a case-by-case approach, whether from the application end, the database end, or a combination.

5.6 Hybrid Database Modelling

At the beginning of this chapter, the statement was made that hybrid modelling may entail a hybrid approach to modelling even when using single database technology. However, hybrid models based on more than one technology should also be discussed as it is just an extension of the same approach.

It was demonstrated that the same modelling techniques and principles apply to SQL and NoSQL models. For this reason, when considering hybrid database approaches, one only has to determine whether an entity exists *only* in one database or more than one.

When an entity depends on another entity that exists only in another database, the application has to handle joining data and ensure consistency. If an entity should be available on both databases (due to specific requirements), some duplication and synchronisation are needed to provide consistency.

Hybrid approaches are only beneficial if a specific function or feature is required or existing architecture has to be integrated into new developments.

Using tools like `mongo_fdw`[14] which allows transactions with Mongo from within Postgresql makes hybrid approaches much easier. It also allows MongoDB to be exposed to a SQL interface.

The following factors, among others, influence the overall cost of hybrid technology designs:

[14] `https://github.com/EnterpriseDB/mongo_fdw`

- Duplication of data;
- Complexity of design;
- Availability due to interdependence;
- Maintainability;
- Flexibility.

Hybrid designs may require additional inserts/splitting models, duplication, and redundancy, but the cost has to be evaluated.

Architecture

The previous sections described ways to model the data better. The following sections will discuss aspects relating to high availability and scalability.

5.6.1 Replication

Replication, the process of duplicating data, happens between a primary and one or more secondary database nodes. It typically comes in two varieties: (i) synchronous and (ii) asynchronous replication.

Synchronous replication delays committing operations until the operation has been applied to a replica(s). This has the benefit of no replication lag, but throughput also suffers [39].

Asynchronous replication, in contrast, replicates in parallel to other operations. Hence it is possible, and likely, to lag behind the primary server. This method does not significantly impact throughput [51].

Lagging implies that if queries were to run on a secondary, there is a margin of staleness (typically not more than a few seconds). The staler a replica becomes, the greater the risk of replication failing due to the replication window growing too small to handle the lag. This is true for MongoDB and PostgreSQL [48, 51].

The purpose of replication is to have a backup of data and the ability to be highly available in the event of a primary failure. When this happens, a secondary can be promoted and assume the role of the primary. It is possible (especially for asynchronous replication) to lose some data equal to the lag window [48].

Managing replicas in MongoDB is very intuitive and user-friendly. On the other hand, PostgreSQL leaves many of the fail-over functions for the user to implement. Many open source and free tools exist to manage replication for PostgreSQL, such as `replmgr`[15] and

[15] `https://repmgr.org/`

`EnterpriseDB Failover Manager`[16].

5.6.2 Partitioning

Partitioning is the ability of a table to be partitioned either horizontally (by splitting rows) or vertically (by separating columns). Each partition can be stored on a separate physical disk, thus increasing read/write speed, optimally making use of cache and decreasing the search surface [51, 54].

MongoDB cannot partition traditionally but instead adopts the concept of sharding, which closely resembles horizontal partitioning, but automatically and dynamically rebalances partitions to have equal (or deterministic) splits [43, 48].

Sharding

Sharding allows for the horizontal scaling of reads and writes. Both databases can partition (or shard) based on a range of keys or hash. Sharding, in addition to regular partitioning, migrates a partition to a different server altogether. Not only is disk performance scaled, but also reads, writes and CPU-bound work [48, 50]. MongoDB has built-in support for automatic and dynamic sharding and rebalancing [48]. PostgreSQL can be sharded manually by combining partitions with foreign data wrappers, or it must use external tools such as the open source Citus[17]. Manually balancing shards can be difficult, complex and time-consuming and is discouraged for large production environments [50, 169].

Employing tools such as Citus, PostgreSQL's automatic sharding capabilities become on par with the offerings inherent in MongoDB.

Sharding can be done on a shard key or a hash key. Shard keys have the advantage that queries will only be sent to matching shards, but it could also lead to hot shards, where queries are not distributed well enough. Choosing a good shard key (or compound shard key) can prevent such problems. Selecting a shard key is similar to building an index, with the specific requirement that the key is to be included in queries that target specific shards [6, 72].

On the other hand, hash keys equally divide a table/collection based on a hash value. Searching by anything unrelated to the hashed key will be spread out to all shards. Writes, however, will be balanced very well. Hash keys are primarily used when there is no good shard key available [6, 72].

Sharding is the only way to scale out both reads and writes, and modelling data so that it can sufficiently be partitioned and distributed is the metric for scalability [54].

[16] `https://www.enterprisedb.com/docs/efm/latest/`
[17] `https://www.citusdata.com/product/community`

Good shard keys, singular or compound, have the following traits, and being able to identify such a key in a collection or table indicates good scalability [48]:

Large cardinality
The keyspace is large, and many unique chunks can be identified.

Low frequency
Keys with high frequency, in contrast, would be stored on the same shard, and scalability would decrease.

Non-Monotonically changing
New inserts end up in the same shard until rebalancing occurs because monotonically increasing keys will always be on the outer edge of the keyspace.

5.6.3 High Availability

Combining sharding for scaling up and replication for redundancy produces highly available robust database clusters. Historically this has been a complex problem to address, and many ways exist to accomplish this.

MongoDB supports high availability as a built-in feature and requires minimal configuration and management to enable and benefit from it [72].

PostgreSQL does not have high availability built-in, and tools, some mentioned already, are recommended for replication and sharding. Additionally, tools such as `HAProxy`[18] and `PgBouncer`[19] can aid during failover and query routing to active servers.

5.7 Process Model Summary

After the processes described above have been satisfactorily completed, the design of the database subsystem is complete. All interfaces and functions interacting with the database are defined and documented; thus, the application could be developed and implemented in parallel.

Many facets of database design have been presented in a logical order. Some aspects, typically for DBAs, have now been presented as an integral part of application design, and this is in stark contrast to typical database application design methodologies.

The sections above contain an abundance of detail and examples. Appendix A presents the DSS in a more summarised, concise format, which should be easier to follow, given a firm comprehension of the abovementioned aspects.

[18] `http://www.haproxy.org/`
[19] `https://www.pgbouncer.org/`

5.7.1 Conclusion

This chapter presented a decision support system that is based on a process model with a specific workflow derived from the SE process. Difficult design choices, decisions, methods and techniques are included to assist with the use of the process model. The process model was derived from an extensive literature study, subject matter expertise, and alignment with best practices.

The overall framework fits in with the research design, where the following links can be made to close the loop between the research design and the DSS model:

The process, item 1 in Figure 2.4 was implemented in the form of the workflow as presented in this chapter and Appendix A.

The process is a model that must be applied to a specific data-based application development in order to become an instantiation of the process model, as is usual in the DSR paradigm.

Procedures and People, item 2 in Figure 2.4 are shown as sub-sections of the workflow. Procedures are embedded in the process and will be executed by database developers and designers.

Specifically, design procedures are present in the Preliminary and Detail design phases, where database developers and designers (respectively) are employed to implement these procedures.

Methods and Techniques, item 3 in Figure 2.4 are evident from the process in places where decisions must be made, quantitative or qualitative, as can be seen in Appendix A.

These techniques are specialised and must be applied based on specific conditions and considerations that are application and technology-dependent.

Technology, item 4 in Figure 2.4 is the selection of a DBMS or a combination of DBMSs (hybrid) to implement the database model developed in the process.

The technology selection is simplified by knowing that the performance of a DBMS is not critical (given similar performance characteristics). Design choices are more important than the effect of design-dependent parameters on the technology selection.

The designer may reference the empirical results from chapter 4 in the design process to best utilise the underlying DBMS. For example, a designer may improve the system's performance by employing multiple threads; similarly, designing for batch operations instead of single operations can improve the application's overall performance. More details on design choices are provided in Appendix A.

This chapter is not an exhaustive guide to design. Nevertheless, the process considers all critical design aspects and presents it in such a way that the underlying database technology becomes less important in terms of performance. Furthermore, the DSS process model informs critical choices, such as whether to embed or reference and when to use indexing.

It is important to emphasise that database design is not linear and is, as such, not a simplistic process. It is imperative to evaluate performance and structure iteratively, to review, and to improve the design in light of the project's full life cycle using a systems perspective. Once all constraints have been adhered to and all the requirements efficiently met, the implementation phase can commence.

When you see validation for a life's work and dedication, it's a beautiful day.

Mary Gauthier

Chapter 6

Use case validation

Introduction

The purpose of the empirical characterisation experiments (Chapter 4) was only to test basic operations and how design parameters affect them. These findings and the DSS were essentially applied to a TPC-C implementation.

TPC-C has many complex and nested queries, spanning multiple tables and many operations and serves as a validation test that complex operational performance can be improved from knowledge emerging from the empirical characterisation.

The research is not complete without validation of the DSS process model of the previous chapter on a benchmark test. It is imperative to show that the DSS approach will result in an improvement when applied to a generic test problem, which was found in the literature as an e-commerce system, called the TPC-C benchmark.

Therefore, this chapter evaluated DBMS performance after following the DSS process on the TPC-C benchmark. The TPC-C benchmark specification defines the database requirements, how to generate the initial data and perform the five transactions used in the benchmark system [152]. To remain objective, both traditional and modern approaches were applied to both PostgreSQL and MongoDB implementations.

Thus four sets of tests were conducted to measure the business performance of the benchmark – the number of *new order* transactions per minute (TpmC) achieved. Source code and instructions are provided in Appendix D.

The research and results discussed in this chapter were published in the 2022 SAIIEE33 conference proceedings [161] and closely correlated with findings from Kamsky who did similar research on MongoDB specifically [156].

The purpose of the use case validation is to use a well-known and documented use case and apply the new knowledge obtained in this research to (i) validate the empirical results in a

real-world use case and (ii) validate that using the DSS can lead to improved performance.

6.1 Background

The TPC-C benchmark consists of the following nine entities, which form part of an e-commerce stock management system [152]:

Warehouse
The warehouse is an entity which can expand to increase the load during benchmarking.

District
Each warehouse consists of exactly ten districts.

Customer
Three thousand customers are populated per district, thus thirty thousand per warehouse.

Order
An order entry stores the customer, warehouse and district. It contains the order's unique ID used to collect all order line items on the order. Orders grow boundless.

Order line
Order lines store the number of items, the item's ID, quantity and amount and a foreign key to the order it is part of. The supplying warehouse ID is also stored. Orders have between five and fifteen order lines.

Item
Items are unique items with a name and base price. Each warehouse has a hundred thousand items.

Stock
Stock are entities which capture the state of an item's stock within a specific warehouse. Each warehouse has a hundred thousand stock items.

History
This entity keeps track of payments made by a client. It stores the foreign keys of the customer, the customer's district and warehouse, and the district and warehouse where the payment was made (if not in the customer's district). History entries grow with each payment.

New order
This table stores new orders by order ID. Each entry is removed upon delivery of the order.

Miscellaneous fields exist in entities to simulate additional data and processing overheads.

Acting on the above list of entities are the following five transactions, categorised by whether they are read-only and the load intensity, ranging from a light- to a heavy-weight impact on performance [152]:

New order
A mid-weight, read-write transaction. An order, order lines and the new order are generated. The warehouse, district, customer and stock entities are modified. It is therefore considered a complex transaction.

Payment
A lightweight, read-write transaction. Only a history entry is inserted while modifying the district, warehouse and customer entities.

Order status
A mid-weight, read-only transaction. This transaction queries a customer's last order's status.

Delivery
A mid-weight, read-write transaction. A batch of ten new (non-delivered) orders is processed. A new-order entry is deleted, and the order and customer entities are modified.

Stock level
A heavy-weight read-only transaction. Stock-level transactions count the number of recently sold items with a stock level lower than a predetermined threshold.

The TPC-C specification describes each transaction in significant detail and provides developers with implementation, verification and execution guidelines [152].

6.1.1 Traditional model

The traditional model is strongly based on the data model provided by the TPC-C specification, as shown in Figure 6.1.

Each entity with a primary key has an index for fast lookups. Additionally, the customer collection has an index for looking up based on the customer's last name, as required on some transactions.

The model is in the third normal form (3NF), as all the fields in each table only refer to the primary key. References to other entities are done solely with foreign keys.

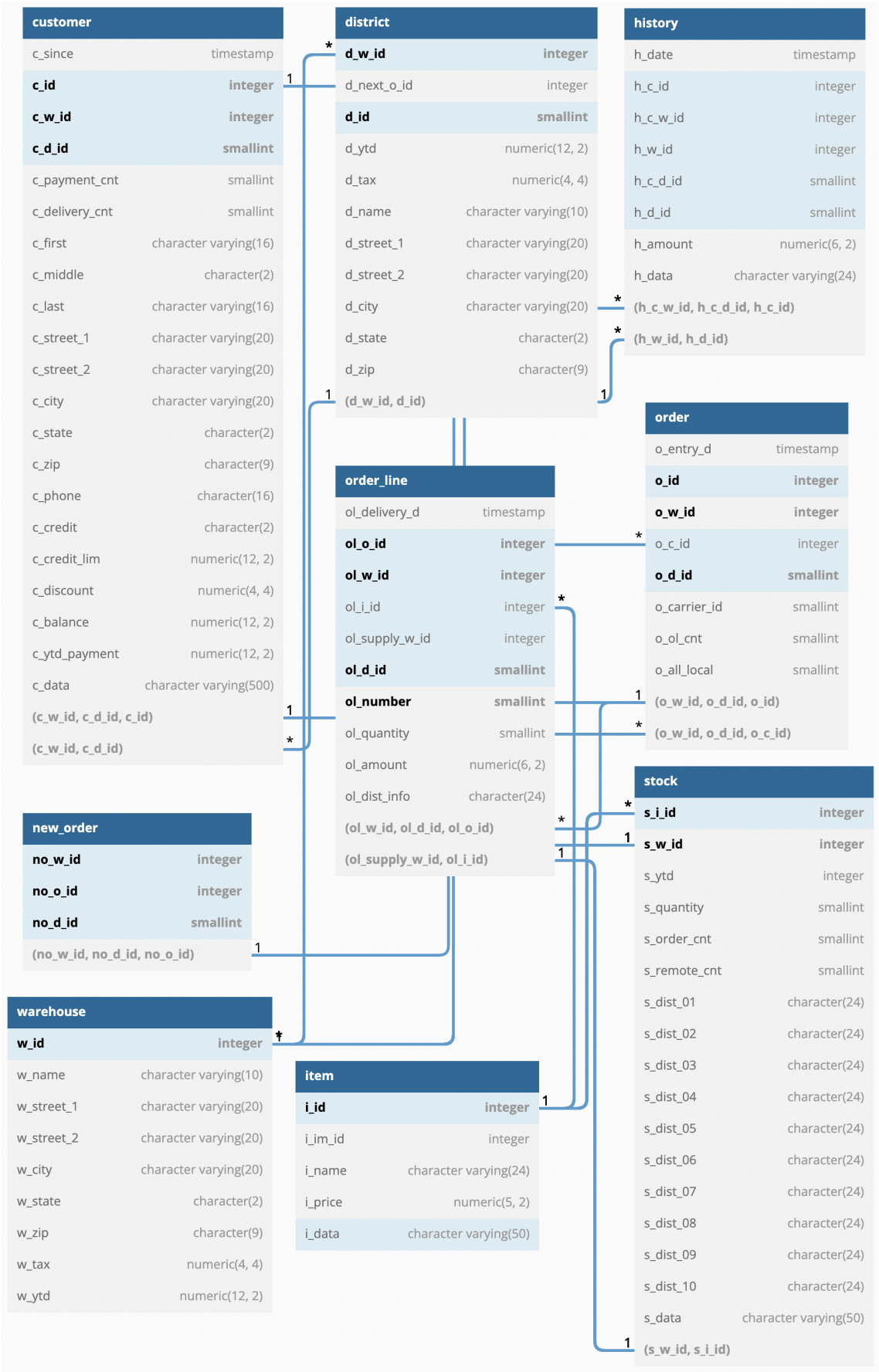


Figure 6.1: TPC-C traditional relational model

6.1.2 Modern model

Considering the relationship between Order and Order Lines, one will find that orders are a *composition* of order lines. Since order lines per order do not grow boundless (only up to ten according to the specification), fully embedding order lines within an order is ideal, as stated in the DSS.

Not only is one table removed, but when querying an order, it only requires a single request to the database. Moreover, common and duplicated fields, such as the delivery date, are shifted into the parent order, reducing the number of write operations required for updating during a delivery. Also, the order's primary key doesn't have to be referenced in the order line items, further reducing duplication, database size and index requirements.

Furthermore, a new order entity is merely a duplication of the primary key of an order. New orders can be represented by a *field* in the orders entity instead of an entire duplicated row in a separate table. This change removes the primary key index of the new order table, and creating a new order has one fewer insert operation.

The final UML class diagram for Orders is shown in 6.2:

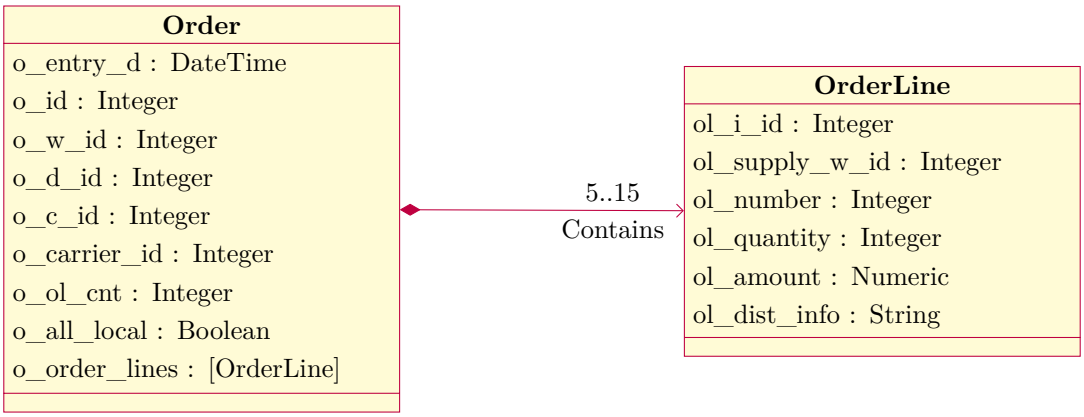


Figure 6.2: UML class diagram for the TPC-C Orders entity

Additionally, one would require an index to look up new orders from the order entities. A partial filter index on the *new* field can be applied to reduce the size of the index.

The new order transaction, therefore, now only requires inserting one document instead of one for the order, one per order line and an additional entry for the new order.

6.2 Experimental setup

The benchmark used the same testing system as the empirical tests to execute the experiments. For each database, a traditional approach was followed during modelling and implementation. The benchmark was run while varying the number of warehouses and the number of concurrent clients, as required by TPC-C.

The same process was followed by the modern approach described in the previous chapter and measuring the TpmC.

6.3 Results

6.3.1 PostgreSQL

Figure 6.3 shows the TPC-C results for PostgreSQL.

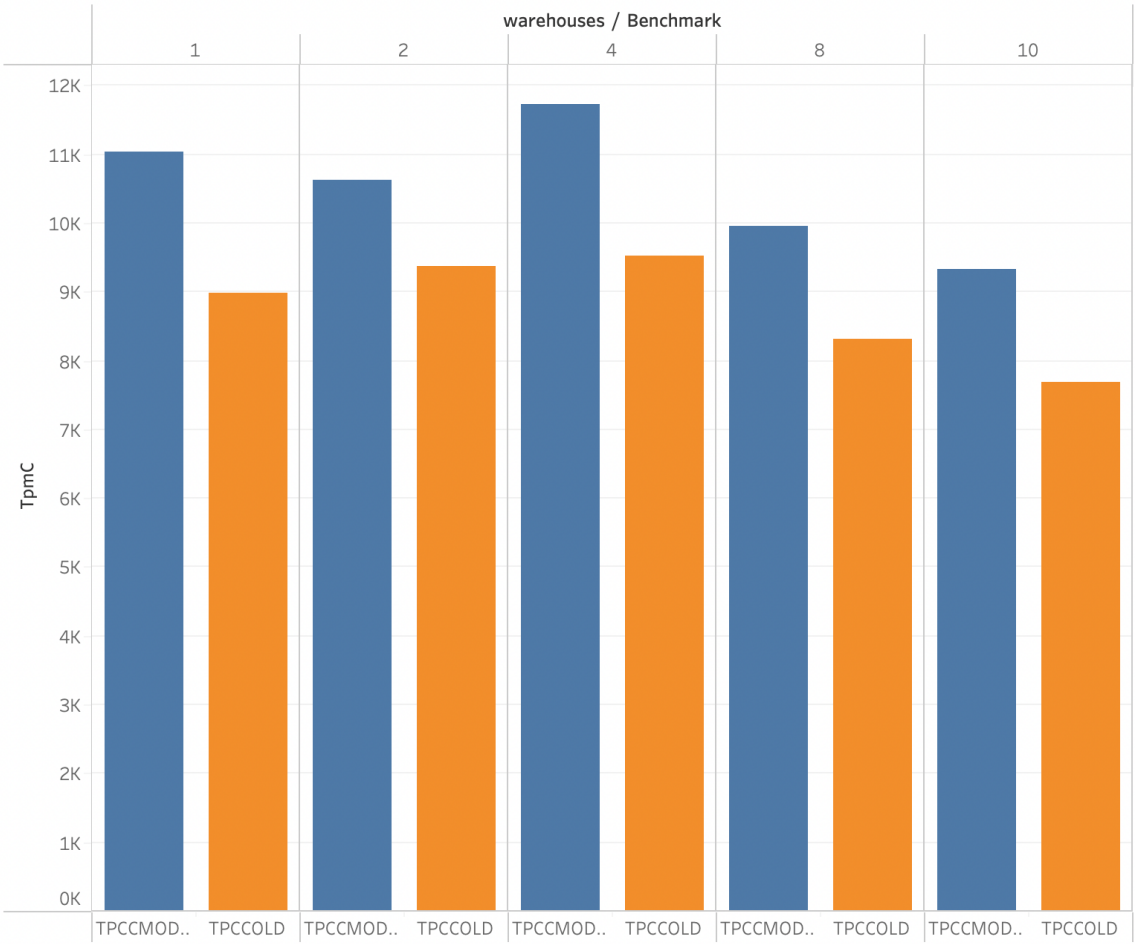


Figure 6.3: TPC-C results for PostgreSQL

An average of 20% improvement in TpmC was found for PostgreSQL when utilizing a modern approach. The results indicate a decline in TpmC for more than four warehouses, which can be attributed to larger working sets which do not fit entirely into RAM.

The impact of multiple threads, or concurrent clients on the system, is shown in Figure 6.4. The empirical results showed that database systems scale well with more threads, which was also true in the benchmark.

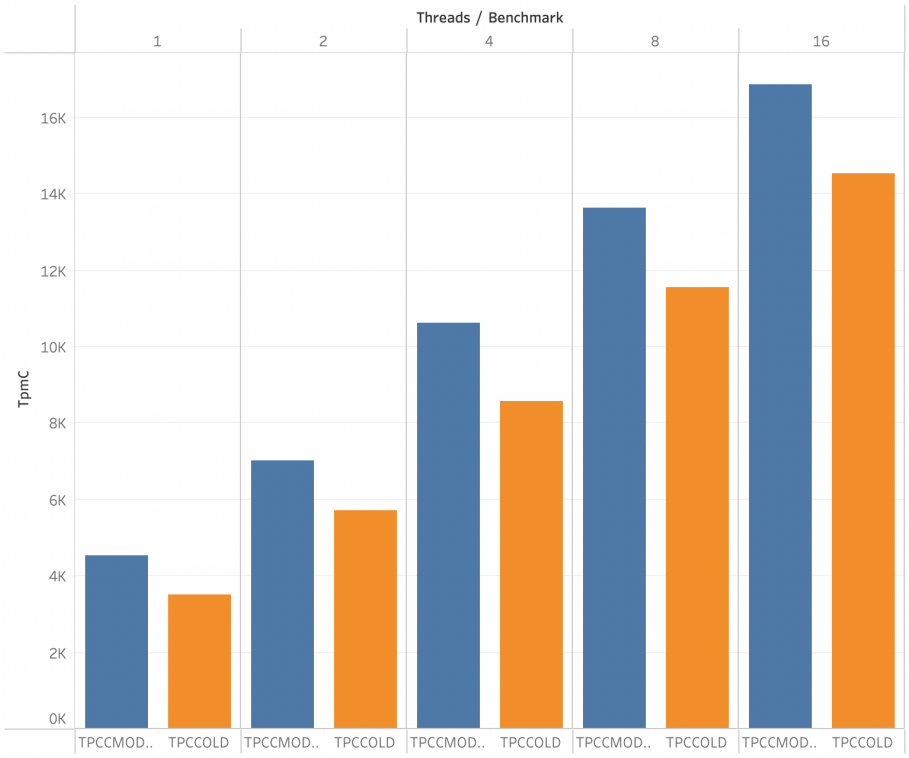


Figure 6.4: Thread impact on TPC-C results for PostgreSQL

6.3.2 MongoDB

Figure 6.5 shows the TPC-C results for MongoDB.

An average of 12% improvement in TpmC was found when employing a modern approach. A similar decline in TpmC was seen when more than four warehouses were used.

The impact of multiple threads, or concurrent clients on the system, is shown in Figure 6.6, which closely correlates with the trends in the results of the PostgreSQL tests.

6.4 Summary

An average increase in TpmC of 20% for PostgreSQL and 12% for MongoDB was found using a modern modelling approach as shown in Figure 6.7.

The improvement can largely be attributed to the reduced database operations and fewer indexes. Even though the modern approach made only a few changes to the underlying model, the model significantly improved the system's performance.

This use case test results validate that a modern approach to data modelling can improve database and application (system) performance for relational and NoSQL database technologies.

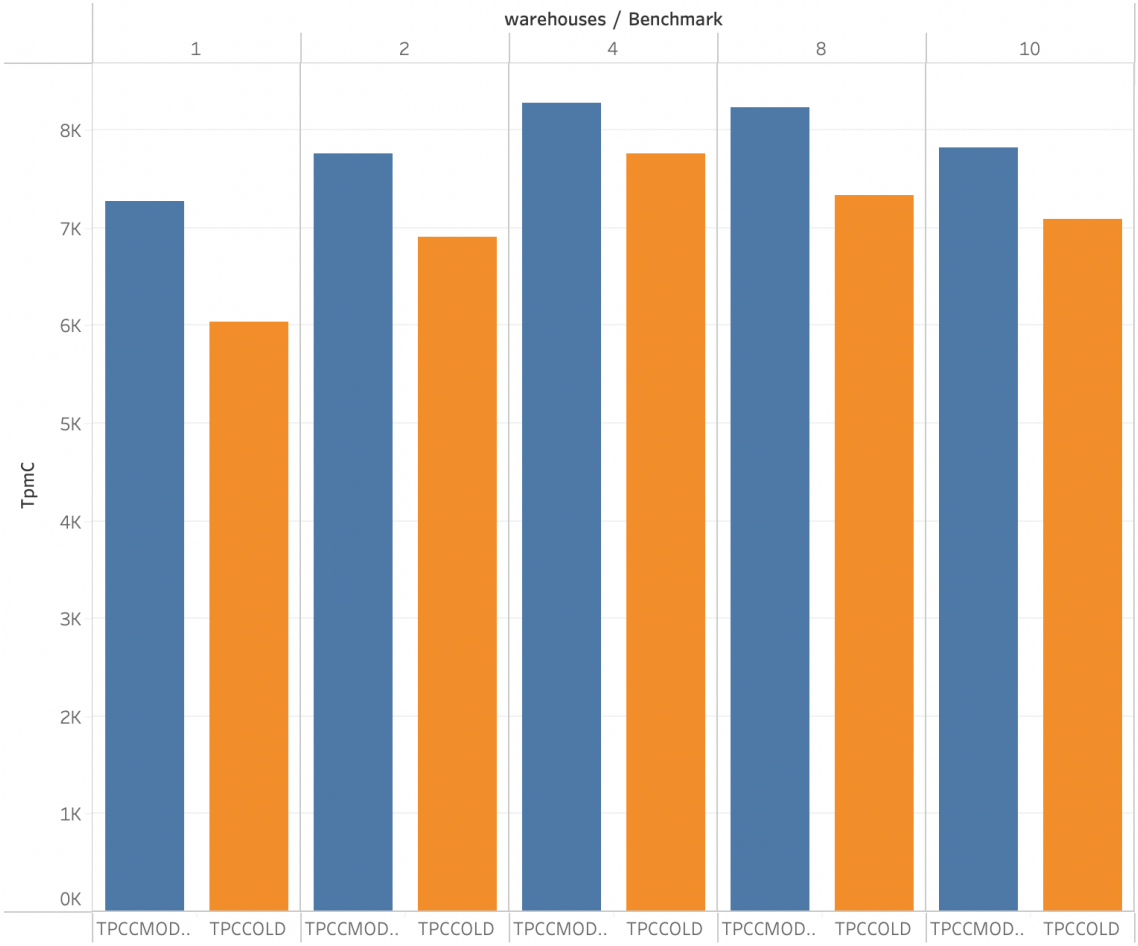


Figure 6.5: TPC-C results for MongoDB

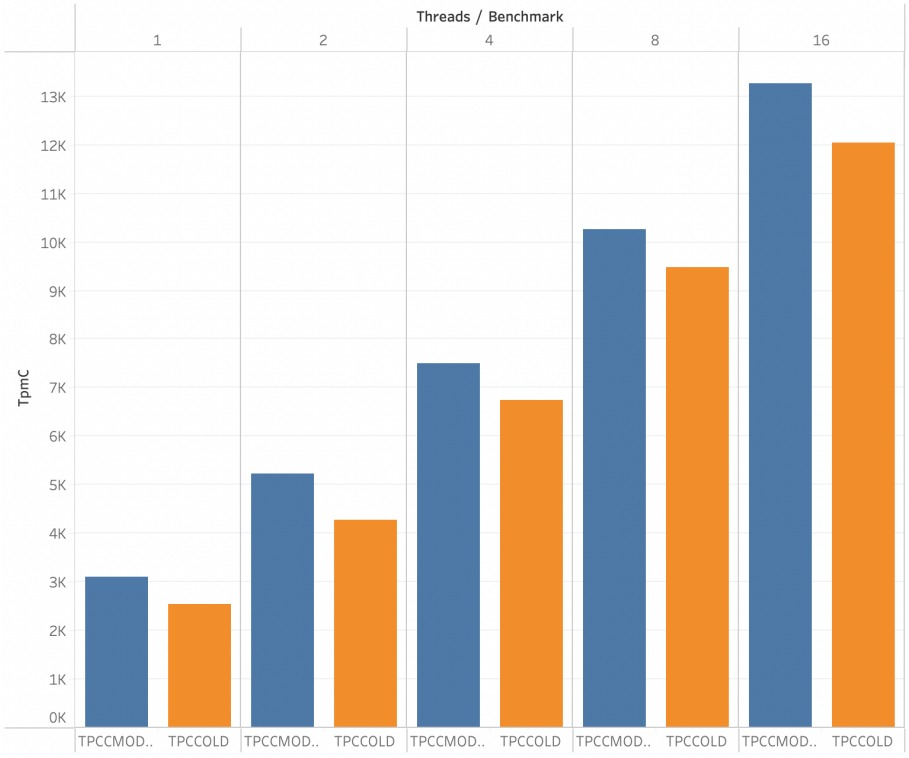


Figure 6.6: Thread impact on TPC-C results for MongoDB

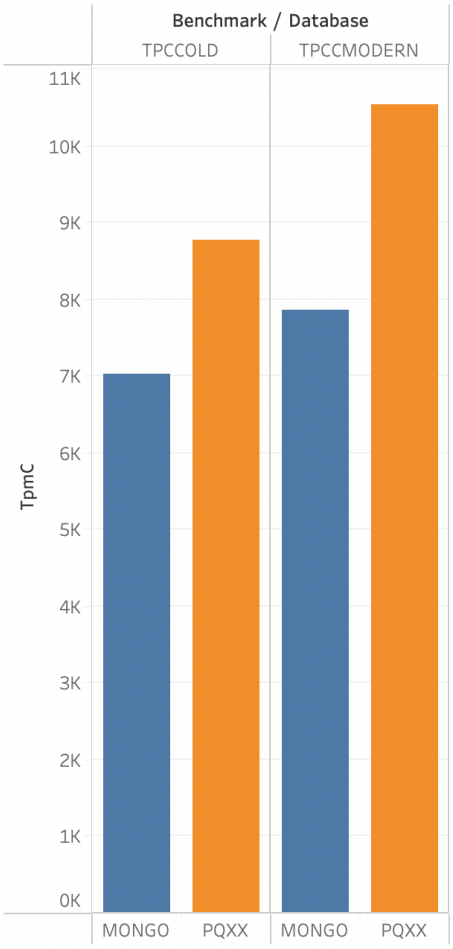


Figure 6.7: TPC-C average results

Furthermore, the guidelines outlined in the decision support system lead to the improved model.

The simpler data model also impacted the complexity of the test applications. In the case of the PostgreSQL implementation, the SQL-based instructions did not easily allow for a hybrid approach for embedding, while MongoDB easily accommodated both modelling approaches.

6.5 Conclusion

The use case validation served two primary purposes: (i) validate the characteristic results in a complex environment and (ii) validate that the DSS can guide a design to have improved performance.

Additionally, this use case also validated the notion that by following the same modern paradigm (from the DSS) on both technologies (PostgreSQL and MongoDB), similar results can be expected.

The performance increase gained from following the DSS process must be considered in the context of contracting the services of a DBMS designer to execute tasks in the preliminary and detail design phases. The actual gain will depend on the size of the application and the cost saving associated with each application. These are highly specific to the environment and the cost structures of a specific organization and must be determined on a "per case" basis.

Considering the research context framework, this chapter contributes to the following, namely that the TPC-C test case validates the DSS process model as follows:

The process, item 1 in Figure 2.4 can identify where optimisation is possible and provides a valid process for the design of a database model from operational requirements.

Procedures and People, item 2 in Figure 2.4 can be applied for the design of a database model by means of procedures that were effective in addressing the requirements of the test case.

Methods and Techniques, item 3 in Figure 2.4 are applicable and pragmatic in a real-world context and a modern paradigm can be applied to relational technology.

Technology, item 4 in Figure 2.4 is not a critical design factor. Similar results were obtained for both relational and non-relational technologies.

In the research project, this chapter validates the findings from the literature study and shows that the process, procedures, and methods apply to a real-world test case. The evidence provided by this validation also confirms the findings from experiments, namely that the choice of technology is less important when a correct process is followed, design choices are made according to the provided guidelines, and implementation is done on both relational and non-relational database technologies. The pragmatic value of the DSS process model is also demonstrated.

Begin thus from the first act,
and proceed; and, in conclusion,
at the ill which thou hast done,
be troubled, and rejoice for the
good.

Pythagoras

Chapter 7

Conclusion

Overview

The Research Validation Matrix, as set out by the Quality Research Management Framework, is primarily used to provide traceability to the research project. Validation and verification of the project occur at various stages in the QRM framework and are illustrated and captured in the RVM.

The completed RVM is presented in Table 7.1 below. Literature topics were condensed to be more concise and not detract from other elements in the RVM.

The following sections will review each of the segments in the RVM and draw conclusions related to the research and contributions thereof.

7.1 Research Problem Validation

In Chapter 2, the original real-world problem context was defined as *How does a real-world data-based application benefit from pragmatic design and implementation?* This real-world problem was then translated into the following research problem: *What are the critical database design and implementation factors that affect database performance?*

To validate that a research problem is, in fact, a valid problem, it is necessary to have sources confirming and defining the need. Studying these sources leads to the identification of a set of research challenges.

The two leading database technologies that must be considered when developing a databases application are relational and NoSQL databases.

It was seen that it is challenging to design data-based applications in a cloud-computing context, and no real pragmatic solution exists that answers all the questions regarding

Table 7.1: Completed RVM

Sources of validation							
Database documentation	↓	↓		↓		↓	↓
Database design literature	↓	↓	↓	↓	↓		↓
Application design literature					↓		↓
Systems Engineering literature					↓	↓	↓
Related work and rationale	↓	↓	↓	↓			↓
Personal observations	↓	↓	↓	↓	↓	↓	↓
Literature Topics \ Concept Research Challenges	No *clear choice* of a database system for data-based applications	*Design and implementation* criteria for each model type are unknown	*Conceptual modelling* approaches are almost exclusively for relational models	No equivalent *logical modelling* approach for both data model types	Application and data model design are usually *separated*	Full *life cycle* approach not considered in methodologies	There is a need for a definitive guide or *strategy* for *designing and planning* data-based applications in the ICT full life cycle
Databases Types	↓↑	↑		↑			↑
Database Design	↓↑	↓↑	↓↑	↓↑	↓↑	↑	↑
Data Modelling		↑	↓↑	↓↑	↓↑	↓↑	↓↑
Cloud Processing							↓↑
Life cycle of an Application	↓↑				↓	↓↑	↓
Economic Factors	↓↑				↓		↓
Human Factors	↓↑				↓		↓
Systems & Systems Engineering	↓↑				↓		↓
Specific Research Solutions \ Concept Research Solutions	Determine strengths and weaknesses of each database	Critically evaluate each model type and propose criteria	Evaluate different conceptual modelling techniques	Use best practices to model equivalently in both types	Use a full life cycle systems approach to data-based application design		Design a decision support framework for database design
Empirical database performance characterisation	↑						
Identification of a common and representative database use case	↑						
Conceptual modelling of application scenarios			↑		↑		
Modelling application scenarios for relational data model(s)			↑	↑	↑		
Modelling application scenarios for non-relational data model(s)			↑	↑	↑		
Critical evaluation of each model	↑	↑			↑		
Design of a decision support framework for hybrid data-based applications		↑		↑	↑		↑

design and implementation. This problem is exasperated by contradicting literature on database technology choices; no clear choice of a database system is identifiable for every problem.

In attempting to design and implement a data-based application, one is confronted with a lack of criteria for each model type, which adds to the complication.

Conceptual and logical Modelling techniques described in the literature are mostly based on methods for relational technology. Investigating NoSQL databases further reveals that there is no equivalent logical modelling approach for both data model types.

The processes involved in application development and database design are typically separated, where the output of database design is a constraint to the application design process. Systems Engineering, in contrast, follows a holistic approach, incorporating all functions and interfaces into the design process. This approach is not prevalent in typical data-based application design in the ICT world.

Upon deeper inspection into the life cycle of applications and database design, from experience and literature, it was seen that when a full life cycle approach is not followed, it leads to problems in the maintenance phase (upstream decisions that impact downstream performance). This is especially evident when requirements are changed post-deployment, often leading to scalability and flexibility challenges.

Considering all the challenges mentioned above, an effective guide or strategy for designing and planning data-based applications in the ICT full life cycle was needed.

Solving the abovementioned challenges should establish the critical database design and implementation factors affecting database performance.

7.2 Research Challenge Validation

After the research problem had been validated, derived research challenges were verified and validated. A literature survey (Chapter 3) was conducted on the following topics (condensed here for simplicity):

Database types A literature survey on the types of databases was done, such as NoSQL, RDBMS and others. MongoDB and PostgreSQL were selected as the representative database technologies due to both being able to model traditionally and with a modern paradigm. However, choosing between these two technologies for a project is still unclear.

Database design Factors that broadly influence database design were researched in the literature study. The literature revealed important design factors, especially concerning cloud contexts, such as the CAP theorem and ER.

Data modelling Modelling is one of the critical processes followed in developing and implementing data-based applications. ER is typically followed for relational models, where entities are strictly in the third normal form. NoSQL modelling is a new research subject, and few sources provide an equivalent modelling strategy.

Furthermore, since NoSQL follows a modern paradigm, it is often difficult to reconcile with traditional mindsets and terminology. Application query patterns' impact on database designs was notably missing in modelling literature, confirming the separation prevalent in data-based application design.

Cloud processing The target environment contributes to many of the design constraints, and cloud computing has specific requirements such as scalability, availability and predictable performance.

Application life cycle From the literature, it was seen that database design and modelling seldom take a full life cycle approach, leading to scalability and flexibility problems when requirements change.

Economic factors Cost of an application development project was seen to be directly related to its duration. It follows that if the development process can be expedited by having a guide to aid in making difficult design decisions, it can reduce costs.

Human factors Human resources require remuneration and directly influence a project's cost. Furthermore, if a project is complex, it adds additional delays, leading to increased project expenditure. Training and education are critical to reducing losses due to complexity.

Moreover, project costs can be reduced if a guide were available to aid important and often time-consuming decision-making.

Systems & Systems Engineering From the research design, SE was identified as a methodology based on a full life cycle approach. Following an SE approach leads to systems with the appropriate form, fit and function. These intrinsic values lead to robust and maintainable systems. It was seen that an overlap between the ER design flow and the SE design phases exists. The overlap mostly coincides with the SE preliminary design phase but with important distinctions.

From the reviewed literature topics, the original research problem and research challenges were verified and validated.

7.3 Concept Research Solution Validation

After thoroughly analysing the literature topics above, conceptual research solutions emerged in the context of the research challenges. The solutions are described below in the context of their corresponding research challenges.

No *clear choice* of a database system for data-based applications: Making an informed decision on which database technology or model to use requires knowledge regarding their strengths and weaknesses. A sufficient conceptual solution was to determine each database's strengths and weaknesses.

***Design and implementation* criteria for each model type are unknown:** When designing or implementing a data-based project, it is required to have criteria by which it can be evaluated. Selecting the correct criteria by which to measure an application's form, fit, and function are essential. Therefore, the concept solution proposed to critically evaluate each model type and propose criteria for evaluating the design and implementation of a data-based application.

***Conceptual modelling* approaches are almost exclusively for relational models:** Making an informed design choice requires a broad review of modelling techniques. Determining their pros and cons and appropriate use cases are essential. The solution required was, therefore, to evaluate different conceptual modelling techniques.

No equivalent *logical modelling* approach for both data model types: Logical modelling is the process of translating the generic conceptual data model into the database-specific format. Obtaining a firm comprehension of the best practices for modelling each specific data model type was critical as a concept solution. This would allow the designer to achieve an equivalent logical model for both data model types.

Application and data model design are usually *separated* & Full *life cycle* approach not considered in methodologies: Addressing the separation of the application and data model design and the lack of a full life cycle approach, a full life cycle systems approach should be followed. SE was identified as a holistic approach to design, incorporating and considering all interfaces and functional units.

There is a need for a definitive guide or *strategy* for *designing and planning* data-based applications in the ICT full life cycle: Since the need for such a guide was verified and validated, designing such a guide would address the need.

The concept research solutions have been verified and validated from the literature and align with the research challenges, as shown above.

7.4 Research Solution Validation

The concept research solutions had to be solved to complete the final research solution. Achieving this required identifying, verifying, and validating specific research solutions. This research project identified seven specific research solutions with two essential functions: (i) to solve the concept solutions and (ii) to jointly complete and validate the final research solution, thereby solving the original research problem.

These seven solutions and how they were implemented are discussed in the following sections.

Empirical database performance characterisation

Virtually all database operations, whether simple or complex, are essentially a combination of the four empirical database operations: create, read, update and delete. Determining the strengths and weaknesses of the two databases required a low-level empirical characterisation of their performance.

Each database was tested with parameters that impact the performance of database operations. These parameters were varied to measure the behaviour of each database's performance. This characterisation was performed and discussed in Chapter 4.

Identification of a common and representative database use case

Evaluating modelling techniques and procedures requires a use case fit to the task. A use case representative of typical database applications is the first criterion. E-commerce is an application field synonymous with database technology. Literature verified that e-commerce applies to both MongoDB and PostgreSQL. However, a specific application was required that is complex and can measure many different aspects of data-based application design.

The TPC-C benchmark provided the ideal and robust specification for an e-commerce application aimed at benchmarking database and application design. Though it is typically used in relational contexts, [156] showed that MongoDB (having transactional capability) could implement such applications. The implementation, tests and results were discussed in Chapter 6.

Conceptual modelling of application scenarios

The empirical experiments (though rudimentary) and use case scenarios had to be modelled conceptually. Techniques from the literature and experience guided the evaluation of conceptual modelling techniques while completing this solution in Chapter 4. The source code of the experiments is provided in Appendix D.

This research used SE as an interdisciplinary framework focused on designing, building, and deploying complex systems. A full life cycle approach to SE involves considering the entire system development process, from concept to retirement. In traditional IT systems, this SE approach is not fully employed, as is evidenced in Figure 3.8 where this difference in approach was highlighted.

The purpose of this research was to demonstrate the above shortfall in the ICT discipline. The result was that a preliminary design phase was specifically added to the life cycle process to allow modelling and design at the intermediary level. This is a critical finding that followed from the full life cycle analysis and the documentary research.

Therefore, employing a SE full life cycle perspective on the application scenarios provided valuable insight which guided the technology-specific solutions.

Modelling application scenarios for relational/non-relational data model(s)

Modelling each scenario in the specific relational/non-relational data model required using best practices for each technology, as gathered from the literature and experience. An iterative process was followed to deliver the best data model. This is described and discussed in Chapter 6.

Also, employing a SE full life cycle approach was essential, as with prior solutions. Specific to the TPC-C benchmark, this led to the relationship classification for Orders/OrderLines as a composition. Being a compositional relationship, it followed from the DSS that embedding was the ideal design choice, eventually leading to significant performance optimisation, as reported in Chapter 6.

Critical evaluation of each model

Models have been designed for many of the other specific solutions. During the iterative processes, the models were evaluated and scrutinised to determine factors contributing to the model's performance.

It was found, in Chapters 4 and 6, that both technologies could model scenarios equivalently and with comparable performance. This led to an essential and novel concept — technology should not dictate the modelling process, but rather that *form follows function*, which should guide the modelling process.

Design of a decision support framework for hybrid data-based applications

The decision support system was carefully designed to be pragmatic, incorporating results from a comprehensive literature study (Chapter 2) and empirical experimentation (Chapter 4). SE processes form the backbone of the DSS, leading to a holistic and pragmatic guide. Functional units and interfaces from the whole system are considered in the design process, discussed in detail in Chapter 5.

Furthermore, critical factors that impact database performance, derived from the literature (Chapter 2), experimentation (Chapter 4) and use case testing (Chapters 4 and 6), have been inducted into the DSS.

Decisions, often complex and obscured by traditional paradigms, are addressed by the DSS, which offers a pragmatic *choice* where possible. A summary of the flow of the DSS

is provided in Appendix A.

The SE concept that form follows function, which also emerged from the research in Chapter 4, confirms that technology choices should not be made prematurely. Moreover, by deferring technology choices until after the preliminary design phase, a more generic and pragmatic process followed, as was experienced during the use case test in Chapter 6.

In the NoSQL context, this novelty proved that MongoDB could function well in environments typically assigned to relational databases, such as TPC-C. Similarly, it was shown that PostgreSQL could be modelled with a modern (denormalized) paradigm, resulting in improved performance (as shown in Chapter 6).

The DSS was specifically validated as a pragmatic guide to modern data modelling by evaluating the impact of the DSS on a representational use case (TPC-C), in both a PostgreSQL (relational) and MongoDB (non-relational) context.

Upon reflection, the overall utility of a DSS process in the DBMS development process lies in the predictability (quality), decision support (usability), and design control (structure) that are often lacking in traditional approaches (refer to Chapter 1). The DSS process provides a comprehensive perspective on DBMS development as opposed to a limited ER-type approach. This allows focusing on aspects specific to individual life cycle phases, where the additional emphasis is placed on the preliminary design phase, as was found from this research. As a result, the DSS provides a pragmatic approach to DBMS development as opposed to a technology-bound design approach.

7.5 Final Remarks & Future Work

The research question "How does a real-world data-based application benefit from pragmatic design and implementation?" was answered by (i) a comprehensive and focused literature study, (ii) empirical performance characterisation experiments, (iii) developing the DSS modelling process and (iv) finally validating the DSS against a use-case test.

After documentary and experimental research, and after the rigor cycle in the DSR paradigm, the proposed method required integration into the real world. A representative real-world test case was selected in the form of TPC-C. This is a validated test case against which a method and technology can be benchmarked, which was done in this research (Chapter 6). Hence, the effectiveness of this approach (i.e. its *performability*) was demonstrated for a valid test case, which validated the real-world relevance of the DSR paradigm.

Groundwork has been laid for future research and tests on the DSS and the paradigm it presents. It may be applied to other technologies and approaches and evaluated in practice by system designers in a wide range of use cases. What has been added from an epistemological approach lies in the findings from this research, namely that *the technology*

selection is less important than the modelling process in the preliminary design phase.

This thesis focused on the tabular SQL functionality provided by PostgreSQL. Expanding the findings with performance characteristic experimentation based on PostgreSQL's `jsonb` columns would improve the usability of NoSQL-like applications for PostgreSQL, especially concerning usability and complexity, and perhaps base performance.

In the future, empirical experimentation could be expanded to model actual database behaviour. These models could then be used to *quantitatively* predict database behaviour for more complex operations. Predictions on the empirical operations were accurate with close fits for standard mathematical functions, as shown in Chapter 4. Incorporating machine learning might lead to even more accurate predictions for more use cases and database types.

When performance can then be quantitatively predicted, models can be compared without extensive implementation effort from the database designer.

Bibliography

[1] J. Pokorny, "NoSQL databases: a step to database scalability in Web environment," *Proceedings of the International C* Conference on Computer Science and Software Engineering - C3S2E '13*, vol. 9, no. September, pp. 14–22, 2013.

[2] C. Churcher, *Beginning database design.* New York: Apress, 2012.

[3] P. Ponniah, *Database design and development: an essential guide for IT professionals*, 1st ed. New Jersey: John Wiley & Sons, Inc., 2003.

[4] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, no. 4, pp. 287–317, 12 1983.

[5] B. G. Tudorica and C. Bucur, "A comparison between several NoSQL databases with comments and notes," in *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research.* IEEE, 6 2011, pp. 1–5.

[6] K. Chodorow and M. Dirolf, *MongoDB: the definitive guide*, 2nd ed. Sebastopol: O'Reilly, 2013.

[7] C. O. Truica, F. Radulescu, A. Boicea, and I. Bucur, "Performance evaluation for CRUD operations in asynchronously replicated document oriented database," in *Proceedings - 2015 20th International Conference on Control Systems and Computer Science, CSCS 2015*, 2015.

[8] P. Atzeni, F. Bugiotti, L. Cabibbo, and R. Torlone, "Data modeling in the NoSQL world," *Computer Standards & Interfaces*, no. October, pp. 0–1, 10 2016.

[9] G. Zhao, W. Huang, S. Liang, and Y. Tang, "Modeling MongoDB with Relational Model," in *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies.* IEEE, 9 2013, pp. 115–121.

[10] A. Boicea, F. Radulescu, and L. I. Agapin, "MongoDB vs Oracle – Database Comparison," in *2012 Third International Conference on Emerging Intelligent Data and Web Technologies.* IEEE, 9 2012, pp. 330–335.

[11] R. Copeland, *MongoDB Applied Design Patterns*, 1st ed. Sebastopol: O'Reilly, 2013.

[12] E. Meijer, "All your database are belong to us," *Communications of the ACM*, vol. 55, no. 9, p. 54, 9 2012.

[13] C. Golledge, "Bring NoSQL to Your SQL Database: Get the Best of Both Worlds," *Database Trends and Applications*, no. December, pp. 34–37, 12 2013.

[14] Y.-T. Liao, J. Zhou, C.-H. Lu, S.-C. Chen, C.-H. Hsu, W. Chen, M.-F. Jiang, and Y.-C. Chung, "Data adapter for querying and transformation between SQL and NoSQL database," *Future Generation Computer Systems*, vol. 65, pp. 111–121, 2016.

[15] H. Zhang, Y. Wang, and J. Han, "Middleware design for integrating relational database and NOSQL based on data dictionary," in *Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE)*. IEEE, 12 2011, pp. 1469–1472.

[16] A. E. Lotfy, A. I. Saleh, H. A. El-Ghareeb, and H. A. Ali, "A middle layer solution to support ACID properties for NoSQL databases," *Journal of King Saud University - Computer and Information Sciences*, vol. 28, no. 1, pp. 133–145, 2016.

[17] MongoDB.com, "Performance Best Practices: Transactions and Read / Write Concerns," 2020. [Online]. Available: https://www.mongodb.com/blog/post/performance-best-practices-transactions-and-read--write-concerns

[18] A. Kanade and A. Gopal, "A Novel Approach of Hybrid Data Model in MongoDB." *IUP Journal of Computer Sciences*, vol. 9, no. 3, 2015.

[19] M. Villari, A. Celesti, M. Giacobbe, and M. Fazio, "Enriched E-R model to design hybrid database for big data solutions," *Proceedings - IEEE Symposium on Computers and Communications*, vol. 2016-Augus, no. Dl, pp. 163–166, 2016.

[20] A. Badia and D. Lemire, "A call to arms," *ACM SIGMOD Record*, vol. 40, no. 3, p. 61, 11 2011.

[21] A. A. Imam, S. Basri, R. Ahmad, J. Watada, M. T. Gonzlez-Aparicio, and M. A. Almomani, "Data modeling guidelines for NoSQL document-store databases," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 10, pp. 544–555, 2018.

[22] A. A. Imam, S. Basri, R. Ahmad, and M. T. González-Aparicio, "Schema proposition model for NoSQL applications," *Advances in Intelligent Systems and Computing*, vol. 843, pp. 30–39, 2019.

[23] B. S. Blanchard and W. J. Fabrycky, *Systems Engineering and Analysis*, 5th ed. Harlow: Pearson Education Limited, 2014.

[24] SEBoK Editorial Board, *The Guide to the Systems Engineering Body of Knowledge (SEBoK), v. 2.7*, R. Cloutier, Ed. Hoboken, New Jersey: The Trustees of the Stevens Institute of Technology, 2 2022.

[25] INCOSE, *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*, 4th ed. New Jersey: John Wiley & Sons, 2015.

[26] J. E. Holm and G. P. van der Merwe, "Quality research management improves design research effectiveness," *South African Journal of Industrial Engineering*, vol. 30, no. 3, pp. 238–252, 11 2019.

[27] J. Venable and R. Baskerville, "Eating our own Cooking: Toward a More Rigorous Design Science of Research Methods," *ejbrm.com*, vol. 10, no. 2, pp. 89–100, 2012.

[28] G. Goldkuhl, "Design Research in Search for a Paradigm: Pragmatism Is the Answer," in *Practical Aspects of Design Science*, 2012, vol. 1, pp. 84–95. [Online]. Available: http://link.springer.com/10.1007/978-3-642-33681-2_8

[29] A. Hevner and S. Chatterjee, *Design Science Research in Information Systems. Advances in Theory and Practice*, ser. Lecture Notes in Computer Science, K. Peffers, M. Rothenberger, and B. Kuechler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7286.

[30] A. R. Hevner, "A Three Cycle View of Design Science Research," *Scandinavian Journal of Information Systems*, vol. 19, no. 2, pp. 87–92, 2007.

[31] K. B. Misra, *Handbook of Performability Engineering*, K. B. Misra, Ed. London: Springer London, 2008. [Online]. Available: http://link.springer.com/10.1007/978-1-84800-131-2

[32] G. Van Alstyne and R. K. Logan, "DESIGNING FOR EMERGENCE AND INNOVATION: REDESIGNING DESIGN," *Artifact*, vol. 1, no. 2, pp. 120–129, 10 2007. [Online]. Available: http://scholarworks.iu.edu/journals/index.php/artifact/article/view/1360

[33] M. Hammond and J. Wellington, *Research Methods: The Key Concepts*, 1st ed. Routledge: Taylor and Francis, 2013.

[34] R. K. Yin, *Case study research: design and methods*, 5th ed. Los Angeles: SAGE, 2014.

[35] S. Jacobs, "Systems Engineering," in *Engineering Information Security*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 10 2011, pp. 29–58.

[36] W. T. Chan, "The Role of Systems Thinking in Systems Engineering, Design and Management," *Civil Engineering Dimension*, vol. 17, no. 3, pp. 126–132, 2015.

[37] J. Holtmann, R. Bernijazov, M. Meyer, D. Schmelter, and C. Tschirner, "Integrated and iterative systems engineering and software requirements engineering for technical systems," *Journal of Software: Evolution and Process*, vol. 28, no. 9, pp. 722–743, 9 2016.

[38] Db-Engines, "DB-Engines Ranking," 2019. [Online]. Available: https://db-engines.com/en/ranking

[39] MySQL, "MySQL: An Ideal Choice for The Cloud," 10 2012. [Online]. Available: http://www.mysql.com/why-mysql/white-papers/mysql_wp_choice_for_cloud.php

[40] S. March and V. Storey, "Design science in the information systems discipline: an introduction to the special issue on design science research," *MIS Quarterly*, vol. 32, no. 4, pp. 725–730, 2008.

[41] C. Coronel, S. Morris, and P. Rob, *Database Systems: Design, Implementation, and Management*, 11th ed. Stamford, CT: CENGAGE Learning, 2009.

[42] G. Harrison, "NoSQL and Document-Oriented Databases," *Database Trends and Applications*, vol. 24, no. 4, pp. 32–32, 2010.

[43] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," *IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing - Proceedings*, 2013.

[44] D. G. Chandra, R. Prakash, and S. Lamdharia, "A Study on Cloud Database," in *2012 Fourth International Conference on Computational Intelligence and Communication Networks*. IEEE, 11 2012, pp. 513–519.

[45] M. Rys, "Scalable SQL," *Communications of the ACM*, vol. 54, no. 6, p. 48, 6 2011.

[46] M. Stonebraker and R. Cattell, "10 rules for scalable performance in 'simple operation' datastores," *Communications of the ACM*, vol. 54, no. 6, p. 72, 6 2011.

[47] MongoDB.org, "RDBMS to MongoDB Migration Guide," p. 18, 2015.

[48] MongoDB, "The MongoDB 5.0 Manual," 2021. [Online]. Available: https://docs.mongodb.com/manual/

[49] E. Horowitz, "MongoDB Drops ACID," 2018. [Online]. Available: https://www.mongodb.com/blog/post/multi-document-transactions-in-mongodb

[50] pgDash, "Sharding Your Data With PostgreSQL 11," 2019. [Online]. Available: https://pgdash.io/blog/postgres-11-sharding.html

[51] The PostgreSQL Global Development Group, "PostgreSQL 11.2 Documentation," 2019. [Online]. Available: https://www.postgresql.org/docs/11/index.html

[52] L. Simasatitkul and T. Suwannasart, "A Tool for Generating Relational Database Schema from EER Diagram," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, vol. I, 2012.

[53] M. N. Shirazi, H. C. Kuan, and H. Dolatabadi, "Design Patterns to Enable Data Portability between Clouds' Databases," in *2012 12th International Conference on Computational Science and Its Applications*. IEEE, 6 2012, pp. 117–120.

[54] B. Schwartz, P. Zaitsev, and V. Tkachenko, *High Performance MySQL: Optimization, Backups, and Replication*, 3rd ed. Sebastopol: O'Reilly Media, Inc., 2012.

[55] K. K.-Y. K. Lee, W. W.-C. Tang, and K. K.-S. Choi, "Alternatives to relational database: Comparison of NoSQL and XML approaches for clinical data storage." *Computer methods and programs in biomedicine*, vol. 110, no. 1, pp. 99–109, 11 2012.

[56] M. Sharma and G. Singh, "Analysis of Joins and Semi-joins in Centralized and Distributed Database Queries," in *2012 International Conference on Computing Sciences*. IEEE, 9 2012, pp. 15–20.

[57] D. Pritchett, "Base: an Acid Alternative," *Queue*, vol. 6, no. 3, pp. 48–55, 5 2008.

[58] Y.-l. Choi, W.-S. Jeon, and S.-H. Yoon, "Improving Database System Performance by Applying NoSQL," *Journal of Information Processing Systems*, vol. 10, no. 3, pp. 355–364, 9 2014.

[59] C. Bazar and C. S. Iosif, "The Transition from RDBMS to NoSQL. A Comparative Analysis of Three Popular Non-Relational Solutions: Cassandra, MongoDB and Couchbase," *Database Systems Journal*, vol. V, no. 2, pp. 49–59, 2014.

[60] E. Elias, "Massively Sharded MySQL Tumblr's Size and Growth," 2011. [Online]. Available: http://conferences.oreilly.com/velocity/velocityeu/public/schedule/detail/21678

[61] MariaDB, "Spider Storage Engine Core Concepts," 2016. [Online]. Available: https://mariadb.com/kb/en/mariadb/spider-storage-engine-core-concepts/

[62] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," in *2011 6th International Conference on Pervasive Computing and Applications*. IEEE, 10 2011, pp. 363–366.

[63] M. Tauro and J. Clarence, "Comparative Study of the New Generation, Agile, Scalable, High Performance NOSQL Databases," *International Journal of Computer Applications*, vol. 48, no. 20, pp. 1–5, 2012.

[64] A. Makris, K. Tserpes, V. Andronikou, and D. Anagnostopoulos, "A Classification of NoSQL Data Stores Based on Key Design Characteristics," *Procedia Computer Science*, vol. 97, pp. 94–103, 2016.

[65] R. Hecht and S. Jablonski, "NoSQL evaluation: A use case oriented survey," in *Proceedings - 2011 International Conference on Cloud and Service Computing, CSC 2011*. IEEE, 12 2011, pp. 336–341.

[66] J. Bhogal and I. Choksi, "Handling Big Data Using NoSQL," *Proceedings - IEEE 29th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2015*, 2015.

[67] K. Grolinger, W. a. Higashino, A. Tiwari, and M. A. Capretz, "Data management in cloud environments: NoSQL and NewSQL data stores," *Journal of Cloud Computing: Advances, Systems and Applications*, 2013.

[68] K. Kaur and R. Rani, "Modeling and querying data in NoSQL databases," *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013*, 2013.

[69] M. Indrawan-Santiago, "Database Research: Are We at a Crossroad? Reflection on NoSQL," in *2012 15th International Conference on Network-Based Information Systems*. IEEE, 9 2012, pp. 45–51.

[70] Y. Liu, Y. Wang, and Y. Jin, "Research on the improvement of MongoDB Auto-Sharding in cloud environment," in *2012 7th International Conference on Computer Science & Education (ICCSE)*, no. Iccse. IEEE, 7 2012, pp. 851–854.

[71] MongoDB.org, "MongoDB and MySQL Compared," 2016. [Online]. Available: https://www.mongodb.com/compare/mongodb-mysql

[72] K. Chodorow, *Scaling MongoDB*, 1st ed. Sebastopol: O'Reilly, 2011.

[73] T. Vajk, P. Feher, K. Fekete, and H. Charaf, "Denormalizing data into schema-free databases," *4th IEEE International Conference on Cognitive Infocommunications, CogInfoCom 2013 - Proceedings*, 2013.

[74] C. A. R. F. O. d. Silva, "Data modeling with NoSQL : how, when and why," Ph.D. dissertation, Universidade do Porto, 7 2010.

[75] W. Golab, M. Rahman, and A. AuYoung, "Eventually consistent: not what you were expecting?" *Communications of the ACM*, vol. 57, no. 3, pp. 38–44, 2014.

[76] N. O'Higgins, *MongoDB & Python*, 1st ed. Sebastopol: O'Reilly, 2011.

[77] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing NoSQL MongoDB to an SQL DB," *Proceedings of the 51st ACM Southeast Conference on - ACMSE '13*, no. April 2013, p. 1, 2013.

[78] MongoDB.org, "Use Cases." [Online]. Available: https://docs.mongodb.com/ecosystem/use-cases/

[79] A. Corbellini, C. Mateos, A. Zunino, D. Godoy, and S. Schiaffino, "Persisting big-data: The NoSQL landscape," *Information Systems*, 2017.

[80] A. Prasad and B. N. Gohil, "A Comparative Study of NoSQL Databases," *International Journal of Advanced Research in Computer Science*, vol. 5, no. 5, pp. 170–176, 2014.

[81] K. Bakshi, "Considerations for big data: Architecture and approach," in *2012 IEEE Aerospace Conference*. IEEE, 3 2012, pp. 1–7.

[82] R. Richardson, "Disambiguating databases," *Communications of the ACM*, vol. 58, no. 1, pp. 54–61, 12 2014.

[83] A. De Lucia, C. Gravino, R. Oliveto, and G. Tortora, "An experimental comparison of ER and UML class diagrams for data modelling," *Empirical Software Engineering*, vol. 15, no. 5, pp. 455–492, 2010.

[84] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36, 3 1976.

[85] C. Fahrner and G. Vossen, "A survey of database design transformations based on the Entity-Relationship model," *Data and Knowledge Engineering*, vol. 15, no. 3, pp. 213–250, 1995.

[86] S. S. Shim, "Guest Editor's Introduction: The CAP Theorem's Growing Impact," *Computer*, vol. 45, no. 2, pp. 21–22, 2 2012.

[87] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, p. 51, 6 2002.

[88] E. Brewer, "CAP twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2 2012.

[89] J. R. Lourenço, B. Cabral, P. Carreiro, M. Vieira, and J. Bernardino, "Choosing the right NoSQL database for the job: a quality attribute evaluation," *Journal of Big Data*, vol. 2, no. 1, p. 18, 2015.

[90] D. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," *Computer*, vol. 45, no. 2, pp. 37–42, 2 2012.

[91] M. Stonebraker, "In search of database consistency," *Communications of the ACM*, no. 1, pp. 2–4, 2010.

[92] S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," *Computer*, vol. 45, no. 2, pp. 30–36, 2 2012.

[93] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "HAT, not CAP: Highly Available Transactions," *eprint arXiv:1302.0309*, 2 2013.

[94] P. Bailis and A. Ghodsi, "Eventual consistency today," *Communications of the ACM*, vol. 56, no. 5, p. 55, 5 2013.

[95] E. Eessaar and M. Soobik, "A decision support method for evaluating database designs," *Computer Science and Information Systems*, vol. 9, no. 1, pp. 81–106, 2012.

[96] V. Dinu and P. Nadkarni, "Guidelines for the effective use of entity-attribute-value modeling for biomedical databases," *International Journal of Medical Informatics*, vol. 76, no. 11-12, pp. 769–779, 2007.

[97] G. Sanders and S. S. S. Shin, "Denormalization effects on performance of RDBMS," *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, vol. 00, no. c, pp. 1–9, 2001.

[98] M. Klems, D. Bermbach, and R. Weinert, "A Runtime Quality Measurement Framework for Cloud Database Service Systems," in *2012 Eighth International Conference on the Quality of Information and Communications Technology*. IEEE, 9 2012, pp. 38–46.

[99] A. Kanade, A. Gopal, and S. Kanade, "A study of normalization and embedding in MongoDB," *Souvenir of the 2014 IEEE International Advance Computing Conference, IACC 2014*, pp. 416–421, 2014.

[100] G. Zhao, Q. Lin, L. Li, and Z. Li, "Schema conversion model of SQL database to NoSQL," *Proceedings - 2014 9th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2014*, 2014.

[101] J. R. B. Mengelberg, a. H. Negrete, C. De Postgraduados, E. Montecillo, and M. De México, "Data Modeling for Better Performance in a Bulletin Board Application," *Issues in Informing Science and Information Technology*, vol. 7, 2010.

[102] C. H. Lee and Y. L. Zheng, "SQL-To-NoSQL Schema Denormalization and Migration: A Study on Content Management Systems," in *Proceedings - 2015 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2015*, 2016.

[103] M. C. Ruiz, D. Cazorla, D. Pérez, and J. Conejero, "Formal performance evaluation of the Map / Reduce framework within cloud computing," *The Journal of Supercomputing*, vol. 72, no. 8, pp. 3136–3155, 2015.

[104] D. Jiang, A. K. H. Tung, and G. Chen, "MAP-JOIN-REDUCE: Toward Scalable and Efficient Data Analysis on Large Clusters," *IEEE Trans. on Knowl. and Data Eng.*, vol. 23, no. 9, pp. 1299–1311, 2011.

[105] J. Dean and S. Ghemawat, "MapReduce," *Communications of the ACM*, vol. 51, no. 1, p. 107, 1 2008.

[106] E. Barbierato, M. Gribaudo, and M. Iacono, "Performance evaluation of NoSQL big-data applications using multi-formalism models," *Future Generation Computer Systems*, vol. 37, pp. 345–353, 2014.

[107] L. Bonnet, A. Laurent, M. Sala, B. Laurent, and N. Sicard, "Reduce, You Say: What NoSQL Can Do for Data Aggregation and BI in Large Repositories," in *2011 22nd International Workshop on Database and Expert Systems Applications*. IEEE, 8 2011, pp. 483–488.

[108] L. I. Feng, O. O. I. Beng Chin, M. T. ÖZsu, and W. U. Sai, "Distributed Data Management Using MapReduce," *ACM Computing Surveys*, vol. 46, no. 3, pp. 1–42, 2014.

[109] O. Cure, F. Kerdjoudj, M. Lamolle, and D. Faye, "On the Potential Integration of an Ontology-Based Data Access Approach in NoSQL Stores," in *2012 Third International Conference on Emerging Intelligent Data and Web Technologies*. IEEE, 9 2012, pp. 166–173.

[110] T. Kudo, M. Ishino, K. Saotome, and N. Kataoka, "A Proposal of Transaction Processing Method for MongoDB," *Procedia Computer Science*, vol. 96, pp. 801–810, 2016.

[111] L. A. H. d. S. Maciel and C. M. Hirata, "Fault-tolerant timestamp-based two-phase commit protocol for RESTful services," *Software: Practice and Experience*, vol. 43, no. 12, pp. 1459–1488, 12 2013.

[112] X. Li, Z. Ma, and H. Chen, "QODM: A query-oriented data modeling approach for NoSQL databases," *Advanced Research and Technology in Industry Applications (WARTIA), 2014 IEEE Workshop on*, pp. 338–345, 2014.

[113] P. Deitel and H. Deitel, *C++ How to Program*, 6th ed. New Jersey: Prentice Hall, 2008.

[114] D. Brdjanin and S. Maric, "An Example of Use-Case-driven Conceptual Design of Relational Database," in *EUROCON 2007 - The International Conference on "Computer as a Tool"*. IEEE, 2007, pp. 538–545.

[115] ——, "An approach to automated conceptual database design based on the IML activity diagram," *Computer Science and Information Systems*, vol. 9, no. 1, pp. 249–283, 2012.

[116] S. Warnars, "Object-oriented modelling with unified modelling language 2.0 for simple software application based on agile methodology," *Behaviour & Information Technology*, vol. 30, no. 3, p. 15, 2010.

[117] I. Antovic, S. Vlajic, M. Milic, D. Savic, and V. Stanojevic, "Model and software tool for automatic generation of user interface based on use case and data model," *IET Software*, vol. 6, no. 6, p. 559, 2012. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/iet-sen.2011.0060

[118] Object Management Group, "OMG Unified Modeling Language TM (OMG UML)," p. 794, 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5/

[119] Y. Li, P. Gu, and C. Zhang, "Transforming UML class diagrams into HBase based on meta-model," in *2014 International Conference on Information Science, Electronics and Electrical Engineering*. IEEE, 4 2014, pp. 720–724.

[120] M. Yusufu, H. J. Zhang, G. Yusufu, Z. D. Liu, P. Cheng, and D. Dilisati, "Modeling and Analysis of Complex System with UML: A Case Study," *Applied Mechanics and Materials*, vol. 513-517, pp. 1346–1351, 2014.

[121] K. Al-tahat, "An Innovative Instructional Method for Teaching Object-Oriented Modelling," *The International Arab Journal of Information Technology*, vol. 11, no. 6, pp. 540–550, 2014.

[122] A. Torres, R. Galante, M. S. Pimenta, and A. J. B. Martins, "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications

on application design," *Information and Software Technology*, vol. 82, pp. 1–18, 2 2017.

[123] dbShards, "Scaling Big Data With Your Relational Database," *Database Trends & Applications*, vol. 26, no. 1, pp. 24–24, 3 2012.

[124] T. Jia, X. Zhao, Z. Wang, D. Gong, and G. Ding, "Model transformation and data migration from relational database to MongoDB," *Proceedings - 2016 IEEE International Congress on Big Data, BigData Congress 2016*, pp. 60–67, 2016.

[125] EnterpriseDB, "MongoDB Foreign Data Wrapper Guide," 2022. [Online]. Available: https://www.enterprisedb.com/docs/mongo_data_adapter/5.2.9

[126] P. Ryan and S. Falvey, "Trust in the clouds," *Computer Law & Security Review*, vol. 28, no. 5, pp. 513–521, 10 2012.

[127] K. Ayanlowo, O. Shoewu, and S. Olatinwo, "Conceptual Design and Implementation of a Cloud Computing Platform Paradigm," *... and Intelligent Systems*, vol. 3, no. 12, pp. 41–53, 2012.

[128] T. Kraska and B. Trushkowsky, "The New Database Architectures," *IEEE Internet Computing*, vol. 17, no. 3, pp. 72–75, 5 2013.

[129] A. Bala and I. Chana, "Fault Tolerance-Challenges, Techniques and Implementation in Cloud Computing," *International Journal of Computer Science Issues(IJCSI)*, vol. 9, no. 1, pp. 288–294, 2012.

[130] J. E. Armend´riz-Inigo, I. Arrieta-Salinas, J. Navarro, and A. Climent, "Transactional Support in the Cloud: Taking Advantage of Classic Approaches," in *2011 22nd International Workshop on Database and Expert Systems Applications*. IEEE, 8 2011, pp. 44–48.

[131] S. R. Schach, *Object-Oriented & Classical Software Engineering*, 7th ed. New York: McGraw-Hill, 2007.

[132] S. W. Ambler and M. Holitza, *Agile For Dummies, IBM Limited Edition*, 1st ed. New Jersey: John Wiley & Sons, Inc., 2012.

[133] P. Helland, "If You Have Too Much Data, then "Good Enough" Is Good Enough," *Queue*, vol. 9, no. 5, p. 40, 5 2011.

[134] EnterpriseDB, "Using the NoSQL Capabilities in Postgres," 2014. [Online]. Available: http://www.enterprisedb.com/postgres-and-nosql-capabilities

[135] P. W. Jordaan, "Synthesis and evaluation of a data management system for machine-to-machine communication," Ph.D. dissertation, North-West University, 2013. [Online]. Available: http://hdl.handle.net/10394/9073

[136] Y. Liu and Y. Wang, "A Study on Software Development Month Effort," *Journal of Software*, vol. 10, no. 7, pp. 798–804, 2015.

[137] R. Lagerström, L. M. von Würtemberg, H. Holm, and O. Luczak, "Identifying factors affecting software development cost and productivity," *Software Quality Journal*, vol. 20, no. 2, pp. 395–417, 6 2012.

[138] J. Zhan, X. Zhou, and J. Zhao, "Impact of Software Complexity on Development Productivity," *International Journal of Software Engineering and Knowledge Engineering*, vol. 22, no. 08, pp. 1103–1122, 12 2012.

[139] E. Tüzün and B. Tekinerdogan, "Analyzing impact of experience curve on ROI in the software product line adoption process," *Information and Software Technology*, vol. 59, pp. 136–148, 3 2015.

[140] J. M. Patel, "Operational NoSQL Systems: What's New and What's Next?" *Computer*, vol. 49, no. 4, pp. 23–30, 2016.

[141] D. Ganesh Chandra, "BASE analysis of NoSQL database," *Future Generation Computer Systems*, vol. 52, pp. 13–21, 11 2015.

[142] P. T. A. Mai, J. K. Nurminen, and M. Di Francesco, "Cloud databases for internet-of-things data," *Proceedings - 2014 IEEE International Conference on Internet of Things, iThings 2014, 2014 IEEE International Conference on Green Computing and Communications, GreenCom 2014 and 2014 IEEE International Conference on Cyber-Physical-Social Computing, CPS 20*, no. iThings, pp. 117–124, 2014.

[143] MongoDB, "Real-Time Analytics," 2016. [Online]. Available: https://www.mongodb.com/use-cases/real-time-analytics

[144] ——, "Content Management," 2016. [Online]. Available: https://www.mongodb.com/use-cases/content-management

[145] ——, "Metadata and Asset Management," 2016. [Online]. Available: https://docs.mongodb.com/ecosystem/use-cases/metadata-and-asset-management/

[146] ——, "Storing Comments," 2016. [Online]. Available: https://docs.mongodb.com/ecosystem/use-cases/storing-comments/

[147] ——, "Internet of Things," 2016. [Online]. Available: https://www.mongodb.com/use-cases/internet-of-things

[148] A. Botta, W. De Donato, V. Persico, and A. Pescape, "On the integration of cloud computing and internet of things," in *Proceedings - 2014 International Conference on Future Internet of Things and Cloud, FiCloud 2014*, 2014.

[149] MongoDB, "Catalog," 2016. [Online]. Available: https://www.mongodb.com/use-cases/catalog

[150] ——, "Inventory Management," 2016. [Online]. Available: https://docs.mongodb.com/ecosystem/use-cases/inventory-management/

[151] ——, "Category Hierarchy," 2016. [Online]. Available: https://docs.mongodb.com/ecosystem/use-cases/category-hierarchy/

[152] T. P. P. Council, "Tpc-C," *TPC.org*, no. February, 2019. [Online]. Available: http://www.tpc.org/tpcc/

[153] R. N. Avula and C. Zou, "Performance Evaluation of TPC-C Benchmark on Various Cloud Providers," in *2020 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*. IEEE, 10 2020, pp. 0226–0233.

[154] M. El-Hindi, A. Arora, S. Karrer, and C. Binnig, "Towards a Benchmark for Shared Databases [Vision Paper]," *Datenbank-Spektrum*, vol. 22, no. 3, pp. 227–239, 11 2022.

[155] J. Zhai, F. Zhang, Q. Li, W. Chen, and W. Zheng, "Characterizing and optimizing TPC-C workloads on large-scale systems using SSD arrays," *Science China Information Sciences*, vol. 59, no. 9, p. 92104, 9 2016.

[156] A. Kamsky, "Adapting TPC-C Benchmark to measure performance of multi-document transactions in mongo DB," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2254–2262, 2018.

[157] K. Shin, C. Hwang, and H. Jung, "NoSQL Database Design Using UML Conceptual Data Model Based on Peter Chen's Framework," *International Journal of Applied Engineering Research*, vol. 12, no. 5, pp. 632–636, 2017.

[158] M. Piech and R. Marcjan, "NEW APPROACH TO STORING DYNAMIC DATA IN RELATIONAL DATABASES USING JSON," *Computer Science*, vol. 19, no. 1, p. 5, 2018.

[159] MongoDB, "MongoDB and MySQL Compared," 2019. [Online]. Available: https://www.mongodb.com/compare/mongodb-mysql

[160] M. Fotache and D. Cogean, "NoSQL and SQL Databases for Mobile Applications. Case Study: MongoDB versus PostgreSQL," *Informatica Economica*, vol. 17, no. 2/2013, pp. 41–58, 2013.

[161] P. Jordaan and J. Holm, "A Systems Perspective on Database Modelling and Design," in *SAIIE33 Proceedings*, 10 2022, pp. 1002–1018.

[162] P. W. Jordaan and J. E. W. Holm, "Reflection on MongoDB Database Logical and Physical Modeling," in *2019 IEEE AFRICON*, vol. 2019-Septe. IEEE, 9 2019, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/9133972/

[163] R. Osman and W. J. Knottenbelt, "Database system performance evaluation models: A survey," *Performance Evaluation*, vol. 69, no. 10, pp. 471–493, 10 2012.

[164] A. Cheung, O. Arden, S. Madden, and A. C. Myers, "Automatic Partitioning of Database Applications," *Proceedings of the VLDB ...*, vol. 5, no. 11, pp. 1471–1482, 8 2012.

[165] D. Coupal and K. W. Alger, "Building With Patterns: The Outlier Pattern," 2019.

[166] W. Zola, "6 Rules of Thumb for MongoDB Schema Design: Part 3," 2014. [Online]. Available: https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-3

[167] M. H. Wen and V. O. Li, "Form Follows Function: Designing Smart Grid Communication Systems Using a Framework Approach," *IEEE Power and Energy Magazine*, vol. 12, no. 3, pp. 37–43, 2014.

[168] V. Shukla, G. Auriol, and K. W. Hipel, "Multicriteria Decision-Making Methodology for Systems Engineering," *IEEE Systems Journal*, vol. 10, no. 1, pp. 4–14, 3 2016.

[169] Percona, "Autoscaling Databases in Kubernetes for MongoDB, MySQL, and PostgreSQL," 2021. [Online]. Available: https://www.percona.com/blog/2021/06/23/autoscaling-databases-in-kubernetes-for-mongodb-mysql-and-postgresql/

Appendix A

Decision Support System Process Summarised

This appendix presents the DSS in a summarised and concise format. Processes, procedures, techniques and methods are presented in a hierarchical flow-like list. Refer to Chapter 5 in the thesis for more detail on the steps listed below.

Phase 1: Real-world problem/input

- Coincides with SE requirements elicitation.
- **Important**:
  - Determine actors, entities and actions influencing data;
  - Probe for future requirements and growth.

Phase 2: Conceptual design

- Perform SE *system-wide* functional analysis:
  - Identify functional units and interfaces that act on the database subsystem;
  - Determine operational requirements (SE).

Phase 3: Preliminary design (modelling)

Conceptual Modelling

- Perform SE *subsystem* functional analysis;

- Draft UML use case diagram (optional):
  - Annotate frequency of actions (derived from functional analysis);
  - Note overlap between actors.
- Draft UML activity/swimlane diagram (optional):
  - Note interactivity between actors and actions.
- Draft UML class diagram:
  - Generic entities should be used;
  - Basic attributes should be added;
  - No normalisation to be performed (yet);
  - **Important**: classify and annotate each relationship type:
    - Association;
    - Aggregation;
    - Composition;
    - Inheritance.
  - **Important**: classify and annotate each relationship *specific* cardinality:
  - **Important**: classify weak/strong entities:
    - Strong entity means it could exist without its relationships;
    - Weak entity means it could *not* exist without its relationships;
  - **Important**: annotate potential entity growth rate (based on actions):
    - How often are instances added?
    - How often are instances removed?
  - **Important**: annotate potential entity query rate (based on actions):
    - How often are instances read?
    - How often are instances updated?
  - **Important:** note entities used for multi-tenancy discrimination:

Logical Modelling (Refinement) subphase

- Purpose: iteratively "render" and reduce entities and relationships;
- Add all attributes to entities (UML class diagram);
- Add unique ID field to all entities (optional, but recommended);
- Start refinement process:
  - Identify anchor entities (strong and boundless entities);
  - These will remain standalone entities and will not be purely embedded (for existence).

- Render based on relationship classifications:
  - Determine natural directionality per relationship (single or double-ended).
  - Consider cardinality (simplex first):
    - One-to-one: embed;
    - One-to-many:
      - If boundless then embed the *one*-side entity into the many side;
      - else use bucket pattern.
      - One-to-few: fully embed *few*-side into *one*-side
      - Rules of thumb:
      - Composition/inheritance: embedding right into left;
      - Association/aggregation: right references left.
    - Many-to-many:
      - Pure many-to-many: use join entity (always reference);
      - Many-to-few/Few-to-many: array of references/extended references (embedding discouraged due to duplication);
      - Few-to-few: array of references on both ends (embedding discouraged).
  - Complex:
    - Recursively repeat simplex cases;
    - *Hints*:
      - Start with boundless and strongest first;
      - Often changes? Bad for embedding;
      - Often queried? Bad for referencing;
      - Both? Ratio of write/read hint: optimise writes;
- DBMS choice (stakeholders should use MCDM)
  - Tradeoff subjective and qualitative factors, since performance, is mostly on par.

Phase 4: Detail design

- Choose primary keys:
  - Unique;
  - Indexed;
  - May be single/compound.
- If MongoDB, referential integrity lives in the application;
- Indexes (**important**):
  - Only purpose: faster reading and sorting;

  - Caveat: degrades write performance (keep at a minimum);
  - Required if:
    - Documents/rows grow large;
    - Queries are often required (not ad-hoc).
  - B-tree indexes hints (default):
    - Order is important;
    - Rule of thumb: most selective first to least selective (granular).
    - Sort order is important.
  - Use special indexes for special requirements
    - Partial indexes (smaller/limited range/condition);
    - Geospatial;
    - Full-text search;
    - Covered indexes: all fields queried/selected within the index.
- Design patterns are required for special optimisation requirements.
- Views are useful:
  - Pre-stored query/aggregation;
  - Can be materialised and indexed.
- Consistency design:
  - Denormalisation requires to fan out on update/delete or eventual consistency;
  - Transactions can improve consistency regardless of denormalisation status;
  - Two-phase-commit patterns can be used without transactions if needed.
- Hybrid database (technology) modelling:
  - If entity should live only on one DB: join in the application;
  - If entity should live on multiple DBs: duplicate and synchronise;
  - Calculate the impact (duplication, complexity).
- Replication: almost always use async; sync only where required.
- Scalability = partitioning (horizontal) = sharding:
  - Determine shard key:
    - Hash-based: even distributed write, reads scatter/gather (to all shards);
    - Key-base: deterministic, but could lead to hot shards.
  - Determine partition-ability/scalability (key-based):
    - Low cardinality;
    - low frequency;
    - Non-monotonically changing.
- High availability: sharding and replication

Appendix B

Reflection on MongoDB Database Logical and Physical Modeling

This appendix contains the article submitted and published in the 2019 IEEE AFRICON conference proceedings. The research formed part of this thesis.

Reflection on MongoDB Database Logical and Physical Modeling

1st Pieter Willem Jordaan
School for Electrical, Electronic and Computer Engineering
North-West University
Potchefstroom, South Africa
pieterwjordaanpc@gmail.com

2nd Johann Erich Wolfgang Holm
School for Electrical, Electronic and Computer Engineering
North-West University
Potchefstroom, South Africa
johann.holm@nwu.ac.za

***Abstract*—Traditional relational database design uses conceptual, logical, and physical modeling based on Peter Chen's methods. UML (Unified Modeling Language), though being a set of software development tools, is often used to conceptually model data and relationships. This paper presents a methodical approach to logically and physically model data in MongoDB by utilizing UML conceptual modeling aids. Application of data models greatly impacts performance, scalability and flexibility of data systems. Furthermore, application life-cycle and expansion impact long-term decisions made during modeling. This paper takes into consideration application requirements, access patterns and life-cycle during logical and physical modeling in a cloud environment. The flexibility that a NoSQL modeling paradigm embodies makes logical and physical modeling complex, with no hard-and-fast rules. This paper presents logical and physical modeling concepts required during designing database applications with MongoDB.**

***Index Terms*—database, modeling, UML, MongoDB, cloud**

I. Introduction

Entity-relationship modeling methods and tools developed by Peter Chen [1] have become the *de facto* standard for modeling and designing databases. Though designed for and used by relational databases, it is being applied to NoSQL (Not Only SQL) databases to similar effect [2].

NoSQL databases are, however, very different to traditional databases when it comes to modeling, since scalability, consistency and performance greatly depend on the underlying design, especially considering their flexible schema options [3].

Cloud-based development defines application criteria requirements for developers. Additional emphasis is placed on the following requirements specific to cloud systems [4]:

- Ease of use;
- Scalability;
- Availability;
- Security.

Scalability and availability cannot always be predicted due to changing environments and requirements - these factors should be taken into account when designing a database to allow for flexibility, which fit well into NoSQL database systems traits [5].

MongoDB is a NoSQL database capable of running in the cloud. It provides high availability, horizontal scalability and a flexible data structure [6]. MongoDB has full relational capability, which suits traditional entity-relational modeling techniques [7]. Recently MongoDB gained ACID (Atomicity Consistency Isolation Durability) support, which allows for transactions, much like with relational databases [8], [9].

Flexible schemas with no hard-and-fast rules such as the traditional 3NF (third normal form) [10] for designing conceptual and physical models, make NoSQL database modeling complex.

This paper provides a broad, but detailed, overview of database modeling concepts that have a bearing on design choices for NoSQL databases. We provide specific guidelines on how to employ MongoDB's unique feature set. As a case in point, we conceptually, logically and physically model a simplistic use-case, based on Chen's methods. A discussion on the complexity of NoSQL modeling is provided, followed by a conclusion with further research suggestions.

II. Related Work And Contributions

Related studies have contributed to NoSQL modeling research. Shin *et al.* [2] has provided a framework for modeling NoSQL databases generically from a conceptual UML model based on Chen's framework. However, the study does not discuss the physical modeling phase, and has a generic document-based logical model.

An algorithmic approach was developed by Abdelhedi *et al.* [11] to transform conceptual UML class models into physical models for various NoSQL databases. The logical model is used as an intermediate abstraction layer for a subset of features in three NoSQL database types: column-based, document-based and graph-based.

NoAM (NoSQL Abstract Data Model) presented by Atzeni *et al.* [12] offers a database-independent data model for NoSQL databases, also making use of a UML-based conceptual data model.

De Lima *et al.* [3] has published an algorithmic approach to NoSQL logical modeling. Workflow measurements are to be specified along with the conceptual model when developing logical models. Physical modeling and its effects are not demonstrated.

The previously mentioned research publications does not consider application life-cycle with regards to growth, scala-

bility and flexibility. Furthermore, our study does not attempt to generically model multiple NoSQL databases, but rather reflects on factors supporting and leveraging MongoDB and its specific set of features.

III. Database Modeling

A. Unified Modeling Language

UML, originally developed as a set of tools for software engineering, can be used for data modeling. Class diagrams are a superset of ER (Entity-Relationship) diagrams used in Chen's methods [13].

Tools that this paper will use for modeling include use-case diagrams and class diagrams. Use-case diagrams, as in Fig.1, are used to model actors performing activities on systems [14]. This diagram is used to identify the data elements used in conceptual modeling.

Class diagrams represent data elements, attributes, methods, relationships, inheritance and cardinalities as shown in Fig.2. UML stereotypes can be used to add additional information during logical and physical modeling, such as embedding, references and other MongoDB specific features [2].

Composition and aggregation annotations in class diagrams further give an indication of the nature of relationships, leading to either referencing (entities are standalone) or embedding (entities are not independent) [14].

UML was found to provide better support and be more comprehensive than ER (entity-relationship) diagrams at data modeling by De Lucia *et al.* [15]. Since UML could be used to model software classes, documentation could be shared between application and data models.

B. Conceptual Modeling

The ER design approach starts with a need or problem followed by a requirements analysis [16]. From the requirements analysis actors, interactions or activities and some attributes should be known [10]. From this a use case diagram can be drafted [17].

When requirements are satisfactorily determined, the conceptual model can be designed. This model still does not contain database specific annotations, and could be equivalent to a relational database conceptual design. Conceptual designs contain high-level data entities, attributes, relationships and cardinalities [10], [18]. Even during this phase the conceptual design should have foresight and consideration of the full life-cycle, and proper requirements elicitation will significantly contribute to this cause [19].

The following relationship cardinalities are possible [10]:

- One-to-One;
- One-to-Many;
- Many-to-Many.

Cardinalities could also be specific, for example, a department may have between five and fifteen employees [18].

C. MongoDB Logical Modeling

MongoDB is a document-oriented database and data objects have the familiar JSON (JavaScript Object Notation) format, though stored in a more efficient binary format. Documents are contained within collections which in turn belong to a database [20].

Unlike typical relational databases, MongoDB's data structure is not structured as rows and columns, but can have hierarchy - arrays and sub-documents - and often closely resembles data structures used in an application. There are therefore many ways to correctly model data and relationships, but performance characteristics could vary [20].

Traditionally, during logical modeling, the entity-relational conceptual model is decomposed into 3NF as tables and relationships [1], [10]. Since MongoDB does not typically follow a normalization design paradigm, choices about relationships boil down to deciding on:

- Referencing (normalization);
- Embedding (denormalization).

Even within these two options many variations exist [21]. Some of these options are discussed in the following sections, with some advanced patterns to consider during physical modeling.

1) ***Referencing****:* Referencing is the MongoDB equivalent to primary/foreign key relationships in relational databases [9]. A collection can be referenced by including a field referencing a unique attribute of a document in another collection. A special case of this is referencing another document in the same collection, called self-joining [20]. MongoDB does not however automatically ensure referential integrity and relies on application level functionality to achieve that [22].

One factor that is always a certain scenario for referencing is whether the relationship has a high arity. In other words, a distinction has to be made between *many* and *few* [22]. In the case of few, embedding could be considered, otherwise a single document would grow large (maximum of 16 MiB), requiring more RAM and cause slower data transfers [20].

High-arity many-to-many relationships should typically be modeled with a referenced join-collection, much like in a relational table. Again, if either side was few, embedding could be considered [20].

Advantages of referencing include [20]:

- No redundancy;
- Immediate consistency;
- Supports high-arity relationships;
- Flexibility in queries and indexing (physical model);
- Expanded sharding options.

Disadvantages of referencing are [20]:

- Application level joins;
- Depending on query patterns, limited scalability;

2) ***Embedding****:* Embedding, here, refers to the process of adding a sub-document or fields of a conceptual entity wholly and directly into a related document or documents, instead of creating a separate collection and referencing only the primary identifier of related documents [9].

Embedding has the following advantages [20]:

- Application side joins aren't needed;
- Inherent atomic and isolated operations for one-to-*few* relationships exist;
- Data locality is utilized;
- Inheritance is efficiently modeled;
- Faster query performance is available.

Disadvantages of embedding data include [20]:

- Redundant copies are present;
- Eventual consistency is achieved;
- Application level cascading is required for consistency;
- Poor performance is offered for high-arity relationships;
- Less flexibility is available as collections are pre-joined.

3) Summary: Table I summarizes UML class diagram concepts relating to MongoDB's document model.

TABLE I
UML CLASSES TO MONGODB ANALOGY

UML Concept	MongoDB Concept
Class	Collection
Object	Document
Attribute	Field
Association	Embed or reference
Aggregation	Usually reference
Composition	Usually embed
Inheritance	Usually embed

D. MongoDB Physical Modeling

During physical modeling, the database designer considers access-path-independent and access-path-dependent design choices. This includes application access patterns, indexing schemes and sharding keys [1], [2], [11]. MongoDB physical modeling consists of many factors to consider [9]:

- `_id` choice;
- Schema validation;
- Indexing;
- Sharding;
- Replication;
- Consistency level choices;
- Application access patterns;
- MongoDB advanced features.

Many of these factors will also have a bearing on the underlying logical modeling choices, especially indexing and sharding key choices [9].

All documents in MongoDB *must* have an `_id` field that is unique. In fact all collections include a unique index on `_id` by default, which cannot be deleted. The field itself may contain any type, and even mixed types are allowed within the same collection. Since the number of indexes impact write performance [5], it is sensible to make the most out of the index already provided.

MongoDB provides a mechanism to validate operations based on a schema validation document. This allows consistency in data types and value constraints and forms part of physical modeling [9].

Indexes should be used sparingly and accurately. To clarify, each additional index has a negative impact on write speed and memory usage. Indexes are primarily used to improve query (read) performance. Balancing these two factors is critical, but also difficult and there is no hard and fast rule for every situation [9], [20], [22]. Indexing decisions are directly and wholly dependent on application query patterns, which are derived from behavioural patterns as found from requirements elicitation. Important indexing rules of thumb are summarized here [9], [22]:

- Start with the most selective fields that would exactly match a query;
- Add less selective fields in order of selectiveness;
- Add range/sort fields lastly;
- Ensure range/sort fields have appropriate direction;
- Use projection to benefit from covered queries;
- Do use `explain()` to test query performance.

Sharding is MongoDB's way of scaling horizontally - if used correctly it should scale linearly. Sharding works much like indexing, except there is only one index - the shard-key. Data is partitioned according to the shard-key and distributed among shard nodes based on range specifications. Each shard, and its replicas, contain each index, including the shard-key [20], [22].

Queries are routed to applicable shards depending on the shard key. If a query does not specify the shard key prefix, it is distributed to all shards and normal indexes resolve the query. Shard-keys should start with a high cardinality field or fields, followed by a low frequency field to avoid hot-spots. Hash-based sharding is possible too, which has the benefit of evenly distributed writes, but reads are always distributed to all shards [22].

Here too application query patterns play a telling role. Entities that grow boundless, or must be highly scalable (read and write), should be given proper shard-keys to achieve this. Foresight should also guide sharding possibilities for future expansion even if the original requirements does not stipulate that.

MongoDB's replication is asynchronous by design - hence eventual consistency. Replication allows for high availability by providing fail-over when the master has a breakdown. Loss of data can be mitigated by the consistency level chosen per query or transaction, which entails waiting for a number (mostly majority) of replicas to save the transaction. Reading from slaves is allowed and, if eventual consistency is acceptable for reads, greatly improve read scalability [5], [9], [20].

Consistency guarantees are flexible with MongoDB, ranging from typical NoSQL eventual consistency to strong consistency on document level [9] and across collections [8]. Strong consistency is reached when waiting for majority of replicas to acknowledge new writes, and always reading from a master.

Eventual consistency is employed when reading from slaves, or reading from stale embedded documents [20]. Once again

the query patterns and requirements depend strongly on application requirements.

As shown in the series by Coupal *et al.* [21] there are many design patterns - all dependent on the underlying application access patterns, but should be investigated to solve application specific requirements:

Approximation This pattern provides a mechanism to reduce database operations by making approximations of data for which exact and real-time values are not mission critical. Approximations are made on application side and updated as needed or when thresholds are met, thereby reducing high velocity and consistent updates to the database. An example is page view counters - the exact number of views is less important than knowing if there are hundreds or thousands of views.

Attribute When many similar fields exist in a typical document which are of the same context and have to be searchable in a single query, the attribute pattern is very useful. Instead of creating an index on each field, group the fields under a common key and create a single index on the array field. For example a movie may have different release dates depending on the country. Instead of creating a field per country, create a sub-document in an array field named `release_dates` and add an entry per country. A single index on the embedded date field in the array will thus cover every country.

Bucket With the bucket pattern, arrays of smaller quantity embedments are added into a set of documents. When a bucket's embedded array reaches a specified size, 100 for example, a new bucket is created and linked to the other buckets' main node. This allows for higher arity relationships when embedding, improved pagination and additional sharding options [20].

Computed Similar to the approximation pattern, this aims to reduce database operations, in this case more specifically, database reads. When data is rarely updated, but often read or calculated, this pattern makes use of less regular updates, rather than calculating on each read either by updating less often, or updating on writes. Examples are *100 best...* computations. It is not critical to have by-the-minute calculations, but rather once a day, week, month, and so on.

Document versioning This pattern simply uses two collections, one for the current version of documents, and another for archive purposes.

Extended reference A hybrid between referencing and embedding, this pattern embeds only often required information, while referencing the source document for further browsing if needed. An example would be for online orders keeping only the name and shipping address of the buyer instead of all fields.

Outlier When a one-to-many relationship is most often only a one-to-*few* relationship, this pattern can improve the general case, while still providing for outliers. Embedding is preferred, but for outlier cases an extended field is added indicating additional information resides elsewhere. These cases can then be handled specially, rather than slowing down the general case.

Polymorphic Documents that have most fields in common may benefit from this pattern. In essence, this pattern makes use of a single collection to group similar documents together, since most fields are common. Inheritance in UML classes can easily be modeled with this pattern.

Schema versioning In cases where documents have to change form which is not backward compatible, including a schema-version field may mitigate application complexity. Specific queries can target specific versions and handle them differently according to the older schema.

Subset The subset pattern is another hybrid form of embedding and referencing. In order to reduce working set resource requirements, only the necessary data can be embedded, while still existing in another collection. Caches of the latest 10 reviews could be embedded in a book document, for example. If all reviews were to be required, it can be queried from the reviews collection.

Tree and graph Hierarchical data can be stored in MongoDB documents by storing the direct ancestry in an array. It could also be useful to store the immediate parent in a seperate field. Product category fields can benefit from the tree pattern.

Describing all of MongoDB's advanced features is a daunting feat, therefore this paper briefly describes features directly relating to querying [9]:

MapReduce is a popular NoSQL tool for big-data analysis and aggregation. Results can be re-reduced to have an incremental aggregation.

Aggregation Framework provides a query tool for building rich pipelines of queries, such as grouping, projection and array unwinding. Results can be written to sharded collections, indexed and ready for consumption.

Change Streams can be used as triggers at application level, coordinated by the database.

Specialty Indexes provided by MongoDB include full-text search, geographical indexes, time-to-live (self deleting) indexes and partial filter expression indexes (index items based on a predicate).

Tag-aware replication allows configurable and selectable read and write concerns per operation. Tags are used by the application to direct reads to specific tagged replicas. Similarly writes can wait for successful writes to tagged replicas.

Zone-based sharding provides a mechanism to force a specific range in the shard-key to specific shards. This could be used to geographically distribute writes or distribute according to specific hardware.

These tools allow advanced query capabilities which should be exploited to fit the application needs. Advanced features at the database level also lightens the application development burden and complexity.

IV. Example Use Case

A use case is presented to demonstrate the design process more clearly. Consider the following need description:
An application is required to support a library. The application must catalog books and track checkouts. It should also allow readers to submit reviews and ratings when returning books. Overdue books should be emailed monthly to the librarian, along with a summary of the amount of books rented during the month. A bonus feature is to list the top ten readers of the month, by most books read. Eventually more than one library has to be supported in the same application..

When we create a use-case diagram, the diagram in Fig.1 may result. We observe that the following conceptual entities can be expressed in UML as Fig.2: libraries, readers, books and reviews. Readers read many books, and many books are read, or checked out, by readers - a typical many-to-many relationship. The checkout entity is a construct to accommodate the many-to-many relationship. Also, a book is stored in one or more libraries which is also many-to-many.

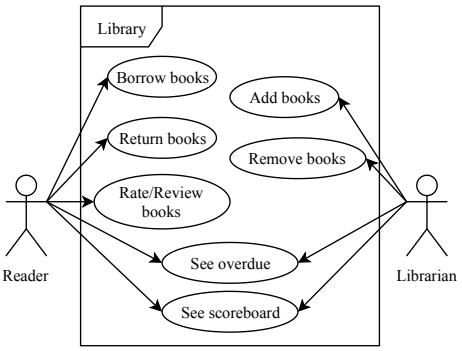


Fig. 1. Use case diagram

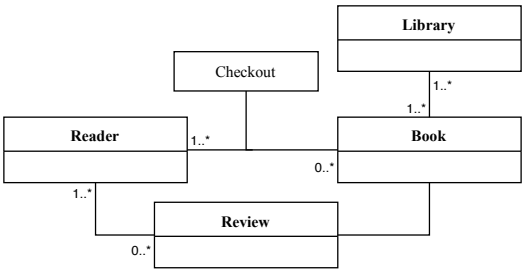


Fig. 2. Library conceptual model

From the description we create an activity diagram showing a typical library visit for a reader in Fig.3. The activity diagram brings context to concepts in the description.

It is defined that the number of books is boundless, but the number of checkouts a reader ever makes is less than 1000. In contrast, the number of times a book has been read (by many people) may exceed 1000, which could happen often. Similarly it is defined that the number of reviews per book could grow well beyond 1000. Books may have more than

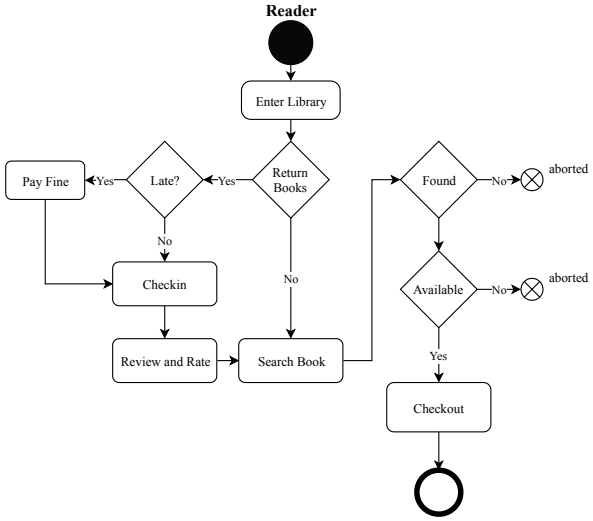


Fig. 3. Reader library activity

one copy in each library. We have decided that 1000 is an appropriate threshold for determining if a concept has to be embedded or not, similar to the example from Coupal *et al.* [23]. The outlier and bucket patterns are to be used for seldom cases that exceed the 1000 threshold.

A myriad of possible designs exists for each concept and a few options are expressed in Table II.

In the light of these constraints and options, libraries and reviews are chosen as collections and checkouts are embedded in the readers collection. In order to support multiple copies in more than one library a LibraryBook concept is required, which links a specific copy to a library - embedded partially in Checkouts and Reviews and fully in Books. Following the modeling process by Shin *et al.* [2] we can now express a document model that supports the requirements as shown in Fig.4.

For physical modeling we need to define application operations, indexes and consistency concerns to allow the queries required: overdue books per reader and books read per reader per month sorted by most reads. Additionally a count of the number of books read must be done. Writes include the adding and removing of books, checking books in and out and creating reviews or ratings.

In order to query overdue books per reader we use a partial filter expression index on the type in `reader.checkouts.checkin` to check if it is `null`. That will however deliver all checked out books, even those that are not overdue. We therefore add the checked out date to the index. Since checkouts are fully embedded it is already grouped by reader. The application just has to determine exactly which of the checkouts are in fact overdue, since the whole reader document will be returned.

An additional index on check-in date allows querying all readers that have checkouts in the previous month. This is done in the aggregation pipeline to unwind the embedded array and

TABLE II
CONCEPTUAL DESIGN OPTIONS

Concept	Referencing/Embedding Design	Pros	Cons
Library	Reference (collection)	Shardable	Joins may be required
	Embed within Book	Data locality	Eventual consistency; difficult to query
	Partially embed within Book	Shardable; data locality	Eventual consistency
Book	Reference (collection)	Shardable	Joins required
	Embed within Library	Data locality; single read operations	Boundless; complex queries;
	Partially embed within Library	Additional feature options	Additional complexity
Book Copies	Reference (collection)	Shardable	Disjoint from Books
	Embed within Library	Data locality	Boundless
	Embed within Book	Data locality; single view on checkout status	Larger working set
Checkout	Reference (collection)	Shardable	Transactions required
	Embed within Reader	Single view; quick access	Transactions required; possibly boundless
	Embed within Reader with bucket pattern	Single view; quick access; not bounded	Transactions required; outlier pattern
	Embed within Book or Book Copies	No transactions	No history
	Partially embed within Book Copies	No transactions	History; eventual consistency
Review	Reference (collection)	Shardable; single full-text index	Joins may be required
	Embed within Checkout	Single review per checkout	Complex queries to group by book/reader
	Embed within Book	Data locality	Boundless
Ratings	Reference (collection)	Shardable; single full-text index	Joins may be required
	Embed within Checkout	Single rating per checkout	Complex queries to group by book/reader
	Embed within Book as counts	Data locality; pre-calculation options	Calculation required per read

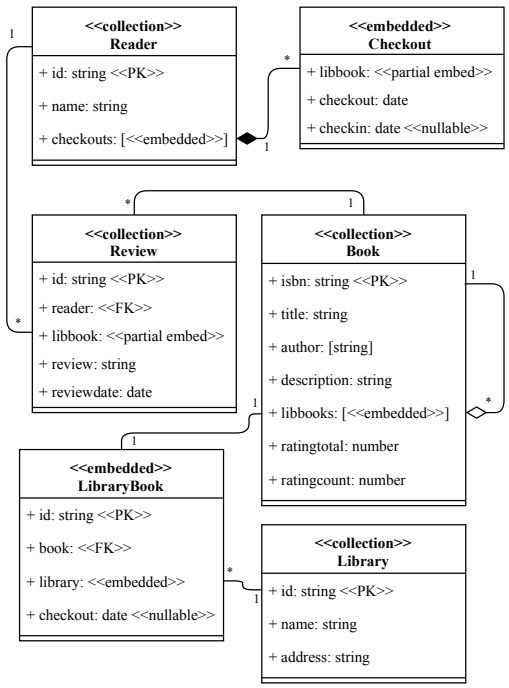


Fig. 4. Library logical model

filter with the same constraints. The resulting list of documents is then grouped and summed by reader. The result is sorted and limited to 10 - the top ten readers of the month.

Counting the number of books rented requires an index on the embedded checkouts' checkout date. Using the aggregation pipeline to match only readers that have checkouts in that month. Unwinding and grouping by library, while summing each checkout will deliver the number of rented books per library.

Average ratings can be calculated on demand by taking the total summed ratings, divided by the count - the fields were already included in the logical model for Book. In order to optimize this, an additional field should be added to pre-calculate the monthly ratings. This can be scheduled daily for example.

Considering the writes to the database, only checkouts are complicated, since a library may have more than one copy of a book and there are more than one library. The LibraryBook embedded array allows for a single book document to keep track of each copy along with an optional checkout date field. If this is set to `null` the book is available. During checkout this is set to the checkout date and when checked in it gets set back to `null`. Similarly in order to keep track of each reader's checkouts and check-ins, a transaction is done to modify the LibraryBook element and modify the reader checkout array.

For this project strong consistency can be employed for all transactions since the few embedded fields, such as library address and book ISBN rarely change.

It is unlikely that sharding would be required for such a project, since reads and writes occur rarely and in person. If needed, though, Book and and Reader can be sharded on primary key.

If it had been required that data per library be isolated a more normalized approach would have had to be followed in order to make the shard key per library. This would

render checkouts, and library books as their own collection. Shard tags could then be employed to isolate specific library information to specific database servers.

In order to keep working sets small it could become necessary to move checkouts to a separate collection, which would require embedding reader information in the checkouts and cascading updates to user details in order for aggregations to group by reader.

Reading from replicas could also improve read scalability, without compromising on consistency for critical operations.

A. Discussion

All design options given in Table II, though not inclusive of every permutation, can validly be modeled in logical an physical form while still meeting requirements. However, each option has an direct impact on the complexity and performance of the application and database design. There is no golden rule, such as the 3NF in relational databases, to aid NoSQL database designers.

Since many design choices rely on application logic, it can be said that the application and database have to be designed together.

Consider, for example, that a reader may only rate a book once. A requirement such as that dramatically impacts design choices for the application *and* database. The current rating system would not be able to support such a requirement.

The complexity of the above use-case could be exponentially increased with the addition of a new concept such as account management. The use-case is not intended to serve as a complete and all-encompassing solution, but instead offer an overview of the complexities involved with NoSQL data modeling due to its inherent flexibility.

Furthermore, specific deployment requirements have not been specified and also have a bearing on underlying design choices, adding an additional layer of complexity.

V. Conclusion

From the example use case it can be seen that MongoDB makes it possible to implement use cases in a variety of ways. How to design a database depends on use case requirements, which must be properly elicited in order to avoid large changes later on. The processes and factors weighing in on the design choices have been discussed in the previous sections and provide a guide to concepts and tradeoffs required during the design phase.

The complexity of designs are affected by the complexity of requirements. Complex designs will inevitably cost more to develop and maintain, though complexity handled during the design phases may result in simpler and more robust solutions in the future.

Form-follows-function [24] - this common term has the implication for database and application design, that function dictates underlying form, structure and design choices. It is the conjecture of the authors that the traditional entity-relational approaches suffer from lack of foresight and understanding of the whole system and its functions and resource requirements. It is instead based on *functions-follow-form* where entities which have data-value are modeled very directly as acted upon by users. Rarely, application access patterns and underlying resources are discussed and weighed in on design choices - especially when considering full life-cycle requirements.

More specifically, future growth is often not accounted for as it is usually addressed in the physical model as a means of optimizing queries as dictated by the structure of the logical design. As shown in the use-case example the physical model design is greatly impacted by and reliant on the prior designed logical model. Furthermore, changes required in the physical model requires changes on the underlying logical model - they are clearly interdependent.

This paper therefore submits that physical and logical modeling phases in NoSQL, and more specifically MongoDB, should coincide as a single detail design phase. This is due to the fact that choices on either side are dependent on the other. When considered as a unit, it may lead to shorter design phases and less *back-to-the-drawing-board* situations.

In large projects, the requirement for an interactive design effort (physical and logical) implies good communication and coordination between design teams, should such teams be used.

Furthermore, requirements elicitation should provide insight into potential growth and outlier query patterns and requirements, which often impact the general case. The requirements define the functions and constraints, which in turn dictates the form that the application and database (the resources) should take on. It is an outside-in approach taking into consideration the application design and user access patterns *chiefly*.

This is in contrast with applications of Chen's methods, which start off with data-yielding entities, leading to concepts, logical forms and ultimately a physical database. It is typically only *after* this where application access patterns are typically fitted to pre-existing underlying data structures. This is an inside-out data-oriented approach.

Even though the example use case was imagined and artificial, it has conveyed the type of design choices and methods that should be followed for MongoDB applications.

Further research is needed to formally design a database *and* application framework for modeling. It should address the following problems and considerations:

- User activity workflows;
- Impact of requirements;
- Maintenance workflow impacts;
- Design and maintenance costs;
- Deployment impacts.

In general, NoSQL modeling techniques are under-researched and due to their flexible nature, may bring forth new design approaches previously impossible or too complex to attempt with only rows and columns.

Finally, with cloud-based applications usually being the target for NoSQL development, the impacts of its constraints and features on design choices should be further investigated.

VI. Authors and Affiliations

Pieter Jordaan (MEng) is currently pursuing his PhD in Engineering at the North-West University in South Africa. He is a student member of IEEE. His research field includes topics on cloud applications, NoSQL database development and scalability.

Johann Holm (PhD, PrEng) is an associate professor at the School for Electrical, Electronic and Computer Engineering of the North-West University in South Africa. He is a member of IEEE and a member of INCOSE, and is actively involved in research and development, as well as consulting in operations and product development.

References

[1] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36, 3 1976.

[2] K. Shin, C. Hwang, and H. Jung, "NoSQL Database Design Using UML Conceptual Data Model Based on Peter Chens Framework," *International Journal of Applied Engineering Research*, vol. 12, no. 5, pp. 632–636, 2017.

[3] C. de Lima and R. dos Santos Mello, "A workload-driven logical design approach for NoSQL document databases," in *Proceedings of the 17th International Conference on Information Integration and Web-based Applications &Services - iiWAS '15*. New York, New York, USA: ACM Press, 2015, pp. 1–10.

[4] P. W. Jordaan and J. E. W. Holm, "Implementation and verification of a cloud-based machine-to-machine data management system," in *2013 Africon*. IEEE, 9 2013, pp. 1–5.

[5] K. Chodorow, *Scaling MongoDB*, 1st ed. Sebastopol: O'Reilly, 2011.

[6] A. Kanade, A. Gopal, and S. Kanade, "A study of normalization and embedding in MongoDB," *Souvenir of the 2014 IEEE International Advance Computing Conference, IACC 2014*, pp. 416–421, 2014.

[7] G. Zhao, W. Huang, S. Liang, and Y. Tang, "Modeling MongoDB with relational model," *Proceedings - 4th International Conference on Emerging Intelligent Data and Web Technologies, EIDWT 2013*, 2013.

[8] E. Horowitz, "MongoDB Drops ACID," 2018. [Online]. Available: https://www.mongodb.com/blog/post/multi-document-transactions-in-mongodb

[9] MongoDB, "The MongoDB 4.0 Manual," 2019. [Online]. Available: https://docs.mongodb.com/manual/

[10] C. Coronel, S. Morris, and P. Rob, *Database Systems: Design, Implementation, and Management*, 11th ed. Stamford, CT: CENGAGE Learning, 2009.

[11] F. Abdelhedi, A. A. Brahim, F. Atigui, and G. Zurfluh, "UMLtoNoSQL: Automatic transformation of conceptual schema to NoSQL databases," *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, vol. 2017-Octob, no. 1, pp. 272–279, 2018.

[12] P. Atzeni, F. Bugiotti, L. Cabibbo, and R. Torlone, "Data modeling in the NoSQL world," *Computer Standards & Interfaces*, no. October, pp. 0–1, 10 2016.

[13] M. Yusufu, H. J. Zhang, G. Yusufu, Z. D. Liu, P. Cheng, and D. Dilisati, "Modeling and Analysis of Complex System with UML: A Case Study," *Applied Mechanics and Materials*, vol. 513-517, pp. 1346–1351, 2014.

[14] Object Management Group, "OMG Unified Modeling Language TM (OMG UML)," p. 794, 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5/

[15] A. De Lucia, C. Gravino, R. Oliveto, and G. Tortora, "An experimental comparison of ER and UML class diagrams for data modelling," *Empirical Software Engineering*, vol. 15, no. 5, pp. 455–492, 2010.

[16] C. Fahrner and G. Vossen, "A survey of database design transformations based on the Entity-Relationship model," *Data and Knowledge Engineering*, vol. 15, no. 3, pp. 213–250, 1995.

[17] C. Churcher, *Beginning database design*. New York: Apress, 2012.

[18] P. Ponniah, *Database design and development: an essential guide for IT professionals*. John Wiley & Sons, Inc., 2003.

[19] H. Sammaneh, "Requirements Elicitation with the Existence of Similar Applications: A Conceptual Framework," *2018 International Conference on Computer and Applications, ICCA 2018*, pp. 444–449, 2018.

[20] R. Copeland, *MongoDB Applied Design Patterns*, 1st ed. Sebastopol: O'Reilly, 2013.

[21] D. Coupal and K. W. Alger, "Building with Patterns: A Summary," 2019. [Online]. Available: https://www.mongodb.com/blog/post/building-with-patterns-a-summary

[22] K. Chodorow and M. Dirolf, *MongoDB: the definitive guide*, 2nd ed. Sebastopol: O'Reilly, 2013.

[23] D. Coupal and K. W. Alger, "Building With Patterns: The Outlier Pattern," 2019.

[24] M. H. Wen and V. O. Li, "Form Follows Function: Designing Smart Grid Communication Systems Using a Framework Approach," *IEEE Power and Energy Magazine*, vol. 12, no. 3, pp. 37–43, 2014.

Appendix C

A Systems Perspective on Database Modelling and Design

This appendix contains the article submitted and published in the 2022 SAIIEE33 conference proceedings. The research formed part of this thesis.

A SYSTEMS PERSPECTIVE ON DATABASE MODELLING AND DESIGN

P.W. Jordaan[1*] and J.E.W. Holm[2]
[1]Department of Electrical & Electronic Engineering
Management Cybernetics
North-West University, South Africa
pieter@jerichosystems.co.za

[2]Department of Electrical & Electronic Engineering
Management Cybernetics
North-West University, South Africa
johann.holm@nwu.ac.za

ABSTRACT

The traditional approach for database modelling and design is based on Peter Chen's entity relational methodology. A firm separation exists between application design and database modelling, which leads to challenges as a system's environment and requirements change. The scalability and flexibility of a data model are typically not considered during the design phase due to the disconnect between the processes of the application design and data modelling.

This paper will present an alternative paradigm regarding database and application design, by making use of the systems engineering methodology. A systems approach provides a holistic view of a database system, its functional units and its interfaces. By embedding database design and modelling into the systems paradigm, a generalized approach to data modelling is obtained.

A systems engineering paradigm leads to a closer-knit relationship between application and database development, which allows scalability and agility when an environment changes.

Keywords: Database modelling, systems paradigm, holistic perspective

* Corresponding author

1 INTRODUCTION

Peter Chen established the *de facto* standard for database modelling as early as 1976, when the entity relational (ER) data modelling approach was defined. It is based on the following four data-related tasks [1]:

1. Identification of data entities and relationships of interest;
2. Identification of semantic information, such as relational cardinality;
3. Definition of values and attributes;
4. Organization of data into relations and primary key decisions.

The above tasks form conceptual modelling in the traditional data modelling process. Typically, the artefact of this model is a collection of entity-relationship diagrams or UML class models. Logical modelling is the process of producing a database schema, usually table-based, from the conceptual model. Finally, physical modelling is the process of defining the physical storage structures of a database, mainly automated by database management systems [2].

This process is often uncoupled from application design requirements and primarily considers the data and its relationships. ER could therefore be defined as a middle-out design approach. This separation subsequently leads to difficulty when application requirements cannot be met due to rigid database design methodologies leading to a lack of flexibility and poor scalability. The situation becomes increasingly problematic when a system's environment or requirements change [3].

Considering modern database technologies such as NoSQL, or hybrid approaches, solving the problems mentioned above becomes even more convoluted due to the vast array of modelling and deployment options with no hard-and-fast rules [4], [5].

This article submits that by employing systems engineering (SE) principles with a holistic view of the system, the problems mentioned above become manageable and will produce scalable and flexible systems. In contrast to ER, SE follows a top-down design approach [6].

Figure 1 shows an IDEF0 function diagram. A functional unit consists of inputs, outputs, controls, and resources that govern the function's design and parameters. Inputs could be, for example, system requirements. Resources include human resources, tools, and capabilities. Controls and constraints could be technical or financial, for example [6]. These inputs weigh in on the function and must be considered during its design phase.

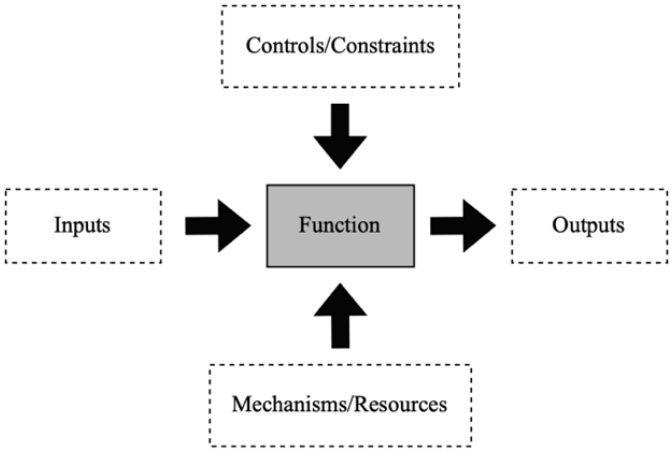


Figure 1: Resource allocation [6]

Moreover, a system consists of many such functions interconnected with interfaces that must be considered during the design, which further advances the need for a holistic database modelling and design paradigm [2], [3], [7].

This article reviews the literature on database modelling and application development *status quo*. Experimental analysis of the impact of modelling parameters on the performance of traditional and modern databases is performed and discussed. These results guide the development of a methodology based on a holistic systems paradigm to aid in the design and modelling of database systems. This methodology, in the form of a decision support framework, is tested against a traditional approach to database modelling and design using a standardised benchmark.

2 LITERATURE REVIEW

This section presents a review of typical database modelling techniques and tasks. Application development and design concepts in a cloud context are presented. Finally, the systems engineering approach is reviewed and applied to database design.

2.1 Database modelling

Typical database design and modelling are based on the following three phases [8], [9]:

- Conceptual modelling;
- Logical modelling;
- Physical modelling.

2.1.1 Conceptual modelling

Traditionally, the first step in database modelling is gathering requirements and developing a specification relating to the database and management system [2].

From the requirements, one elicits a set of entities, their attributes, relationships between them, and actors acting on the database. This elicitation includes cardinalities, that is, how many relations may exist between entity instances [10].

UML class diagrams have become a viable alternative to the traditional ER diagram used to model entities and relationships. Each entity shows its attributes, basic attribute types and relations with basic cardinalities to other entities [11].

Traditionally relationships are grouped into one of the following kinds [12]:

- One-to-one (a person and their spouse);
- One-to-many (a department and its employees);
- Many-to-many (student(s) and class(es)).

Recent developments have provided two additional relationships: one-to-few (less than a few hundred, for example) and one-to-squillions (an unlimited number). These distinctions are essential in modern database development, especially during logical modelling [13].

2.1.2 Logical modelling

After conceptual modelling has been completed, logical modelling transforms the entities and relationships to fit a specific database system, typically a relational table-based database such as PostgreSQL. Logical modelling, traditionally, has the following steps [8]:

- Definition of tables, columns, relationships, and constraints;
- Normalization of tables;
- Ensure integrity constraints are defined.

Tables represent relationships by making use of primary keys and foreign keys. Each entity is assigned a primary key which uniquely identifies that entity. In establishing a relationship between tables, the unique primary key of one entity is placed in another, serving as a reference. Depending on the situation and nature of the relationship, it could be included on

both ends. Foreign keys are therefore used to reference a primary key from a different entity uniquely [8].

Many-to-many relationships are typically substituted with a JOIN table, which serves as a one-to-many and many-to-one mediator between the two sides of the relationship [10].

Normalization, typically the third normal form, organizes data into stable structures to minimize update anomalies and maximize data accessibility. This process is a standard step during traditional logical modelling [14].

Traditional rigid normalization strategies, though supported, have been challenged by the modern NoSQL concept of embedding and denormalization [15].

With the prevalence of NoSQL, and especially document-based databases such as MongoDB, a new paradigm of data modelling has emerged. Tables, or rows of data, have been replaced by collections of documents. The rigidity of columnar, normalized data has been replaced by rich, hierarchical data structures and the ability to have flexible schemas [16].

The divergence from a normalized approach has enabled NoSQL-like approaches to achieve higher scalability and performance, even with traditional technologies [14], [17].

Although the same modelling techniques and principles apply to modern approaches, it is not as clear-cut as the rigid normative approach: improving design flexibility while adversely increasing complexity during the modelling process [16], [18], [19].

2.1.3 Physical modelling

Each database system has specific physical model formats, features, requirements, and constraints [20], [21].

Though indexing, especially concerning integrity, is defined and specified during logical modelling, it manifests in the physical model. Views, partitioning, access control, availability and performance measures are defined in the physical model [8].

Many essential physical modelling tasks can be automated, but performance tuning is frequently delayed until problems arise, often due to poor decisions or lack of foresight during previous modelling phases [2].

2.2 Application design and development

Application development, from the perspective of database roles, happens after database design. Applications interact with databases, though very rarely dictate or influence the design process [2], [10], [22].

Traditionally applications have the following life cycle phases [23], [24], namely, requirements, analysis, design, implementation, post-delivery maintenance, and retirement.

Post-delivery maintenance manifests when specification changes or enhancements are required, often entailing changes in database models or designs [23].

2.2.1 Cloud development

Three specific requirements for cloud applications exist [25], [26]:

- Elasticity – the ability to scale horizontally;
- Flexibility – being adaptable (including data models and schemas);
- Fault tolerance - being highly available.

The underlying database design greatly influences a system's ability to adhere to the above requirements [27], especially when the needs for an application change or schema evolution

takes place. In these cases, schema migration has to occur on relational databases, while NoSQL databases are flexible when changes to a model are required [28], [29].

2.2.2 Unified Modelling Language

Unified modelling language (UML) is often used for software and database development. It is an industry-standard suite of notations, primarily in the object-oriented domain. UML's essential diagram tools support conceptual database modelling and application design, including use-case diagrams, activity diagrams, sequence diagrams and class diagrams [30].

2.3 Systems engineering

Systems engineering, as a methodology, focuses on form, fit, and function. It defines various tools and methods for specifying, designing, creating, operating, and maintaining functions [31].

The systems engineering process is divided into the following two main life cycle phases and activities [6]:

- System acquisition:
  - System design;
  - Implementation;
  - Test and integration;
  - Construction or production;
- System utilization:
  - Operation;
  - Maintenance and support;
  - Upgrade;
  - Retirement.

Figure 2 juxtaposes system development activities (system acquisition phase) in the systems engineering and entity relational design approaches. It is of note that there are similarities between the two approaches.

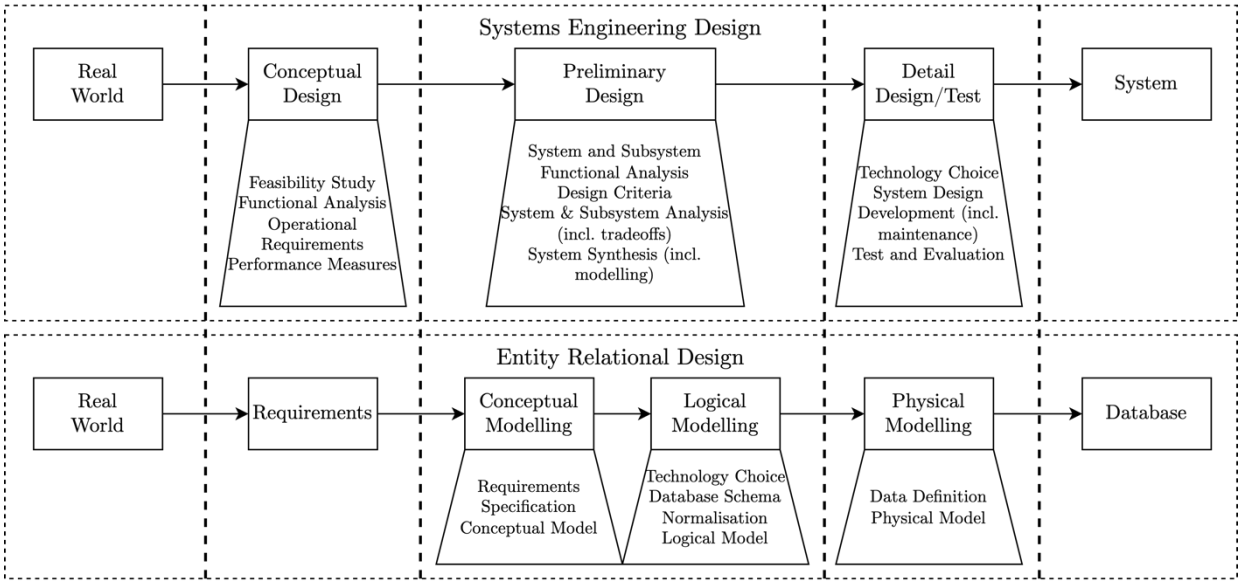


Figure 2: Systems engineering [6] vs entity relational design [2], [22]

A significant distinction is that systems engineering prescribes the modelling activity during the preliminary design phase, whereas ER performs it early on during the conceptual modelling

phase. Furthermore, technology choices are made during the detail design phase, whereas ER chooses a database technology during the logical modelling phase.

Essential to the rationale of this article, the ER process delivers as artefact a documented database, whereas systems engineering delivers a documented functional system, including a database and application as well as all interfaces, subsystems and functions involved [1], [6].

Critically important is that system modelling, evaluation and development specification are the main activities of preliminary design, where the proposed decision support framework delivers the most significant contribution.

3 RESEARCH METHODOLOGY

Two sets of experiments were conducted to solve the problem facing most database designers: Which design considerations are the most important when developing a new data-based application? Two design philosophies must be considered: (i) database technology as the primary determinant or (ii) design methodology as an equalising (technology agnostic) approach.

The first set of experiments focused only on the primary database operations, create, read update, and delete, to determine the performance of two different database technologies not specific to any application test standard or design methodologies. That is, technology options MongoDB and PostgreSQL were objectively compared on a “technology only” test bed.

The second set of experiments evaluates (i) the performance of MongoDB and PostgreSQL after applying a traditional entity relational design process and (ii) the performance of Mongo and PostgreSQL after applying a SE-based design methodology.

The experiments will show that the most significant performance determinant lies in the preliminary design phase (logical database modelling) and not in technology choice. This finding may seem counterintuitive and unexpected, but the experiments followed a systems engineering approach instead of a naïve technology selection process.

Design choices for the second set of experiments were informed by the results from the first set of experiments. Specifically, design parameters such as the number of indexes, fields, and batch sizes were adjusted to obtain optimal performance, regardless of the database technology.

For the second set of experiments, an e-commerce system (based on the TPC-C benchmark [32]) was used as an application benchmark. The benchmark incorporates both the application and database design, and results will therefore holistically reflect the impact of the design methodology on the system’s performance and scalability.

Emphasis is placed on performance engineering because it impacts a system's quality, reliability, maintainability, and sustainability [33].

3.1 First set: Empirical experiments

3.1.1 Purpose

Determine the performance characteristics of two database technologies (MongoDB and PostgreSQL) to identify design parameters for the logical database design activity in the preliminary design phase of the life cycle process.

Create, read, update, and delete operations are empirical functions that are used in all database operations and were used in this set of experiments.

3.1.2 Parameters and setup

The parameters that were varied in experiments include the following: number of read and write indexes, fields projected, stored and queried, number of threads used, the row limit of results, and batch sizes.

The parameter setups for create, read, update, and delete experiments were comprehensively run through relevant permutations. All tests varied the number of threads (from 1 through 8) to measure the local scalability of the database system. Each combination was run on MongoDB 4.4 and PostgreSQL 13 on a controlled commodity machine with default configurations. MongoDB is a popular NoSQL database, while PostgreSQL is a popular relational database with the ability to model complex columns and therefore allows for embedding [20], [21].

To investigate the impact of each parameter, the whole result set was taken into account, and each parameter change was tested for interdependence with other parameters. Each iteration was timed and repeated a sufficient number of times to obtain statistical significance.

3.1.3 Tests

The experimental tests are described below:

1) Test 1 – inserts: This test measures the impact of batch size, number of fields and indexes on insert performance. Documents/rows are generated randomly and inserted based on the parameters.
2) Test 2 – standard queries: This test measures the impact of fields queried, fields returned, fields indexed, and query limit on read performance. The database is pre-populated as described above, and indexes are created or dropped as required by the parameter set.
3) Test 3 – updates: Since updates have both a read and write component, the database is populated with rows of two sets of fields, one used for querying exclusively and one used exclusively for writing. This is done to isolate the impact of each type of index on the measurements. The parameters tested are the number of fields used to query, the number of fields updated, the number of indexes used in a query and the number of unqueried write-only indexes. An increment with random magnitude was used as the operation.
4) Test 4 – deletes: This test measures the impact of fields queried and the number of indexes on delete performance. Like updates, delete operations have a query and a write component, in this case, complete removal of the row/document. The deleted rows are re-inserted after every iteration as it is a necessary step to keep the database pristine for each iteration and set of parameters.

3.1.4 Results and findings

Even though MongoDB generally performed better at write operations and PostgreSQL at read operations, the impacts of the parameters indicated that both database technologies behave in a similar way and with similar magnitudes to changes in these parameters.

In summary, the following observations were made:

1. Indexes significantly improve read performance; exponential gain with > 10^2 on the 6^{th} index
2. Indexes significantly improve sorting performance; exponential gain with > 10^2 on the 6^{th} index
3. Indexes reduce write performance:
    a. Adding more indexes always reduces write performance; logarithmic decay in insert and update tests with between 36% and 50% loss on the 6^{th} index.
    b. If the index helps reduce database scanning, the net effect is positive; exponential gain with > 10^2 on the 6^{th} index

4. High limits reduce the number of operations per second, though inversely increase the number of rows/documents per second;
5. Batching is beneficial for writes in general; logarithmic gain > 10^1 on batches of 4096 versus single inserts
6. Multiple threads/connections are always beneficial; near-linear improvement across all tests (60% improvement per thread)
7. Inserting more fields per row/document decreases write performance; linear decay of around 3%-9% per additional field
8. Projecting more fields reduces read performance; near-linear decay based on the number of rows/documents returned (between 1.5% and 2.5% per field).
9. Updating fewer fields improves write performance; similar behaviour to inserts when unindexed (between 1% and 6% per field) and greater when indexed (between 4% and 12% per field)
10. Querying with fewer fields improves read performance, even when indexed; near-linear decay of 2% per additional field
11. Write operations are far more expensive than reading operations. Ranging from 10^1 to 10^3 times slower for the same number of documents processed.

These findings play a vital role in defining the design choices for the logical database modelling methodology as part of the preliminary design activities. These design choices are typically not applied in traditional database design, and this is because, in traditional database design, strict normalization is the *de facto* philosophy [2].

3.2 Second set: TPC-C experiment

3.2.1 Purpose

This set of experiments aims to evaluate the value of design choices obtained from the first set of experiments in a standard data-based application benchmark (TPC-C).

The TPC-C benchmark is a standard set of functions implemented by a defined set of transactions. Furthermore, the TPC-C benchmark is a standardized database management system and cloud performance benchmark based on complex online-transactional processing (OLTP) workloads. It measures the business throughput: the number of new order transactions processed per minute (TpmC) [32].

The workload is typically modelled for relational databases, but adaptations to fit the benchmark to MongoDB have been successful. The premise of these adaptations is to measure the effect of embedding data rather than relying on strict normalized paradigms [34].

The rationale of using an SE approach is to first characterize the information system, with all its interfaces and interdependencies, before allocating the system requirements to the database subsystem. With the TPC-C benchmark, there is lenience in the design and implementation of the database and application – this was used to demonstrate the effects of allocating system requirements on the database design and performance.

Therefore, this experiment compared (i) a traditional normalized TPC-C application without the value of system requirements analysis and (ii) the SE-based designed database on the TPC-C application after the application of system requirements analysis.

3.2.2 Parameters and setup

The same test system was used as in the first set of experiments. The application was benchmarked against a series of transactions shown below [35]:

- New order – mid-weight read-write transaction;
- Payment – light-weight read-write transaction;

- Order status – mid-weight read-only transaction;
- Delivery – mid-weight read-write transaction;
- Stock level – heavy read-only transaction.

The above transactions were performed and measured (in the application) on both traditional (ER) and SE design paradigms for PostgreSQL and MongoDB to validate that a SE design approach on both technologies scales and performs better, even for more complex transactions. The results were validated against the MongoDB results in [34].

3.2.3 Results and findings

The results of the PostgreSQL experiment are shown in

Figure 3 and MongoDB in Figure 4. An average increase of 20% in TpmC (new-order transactions per minute) for the modern approach on PostgreSQL and 15% for MongoDB correlates with the results found in [34].

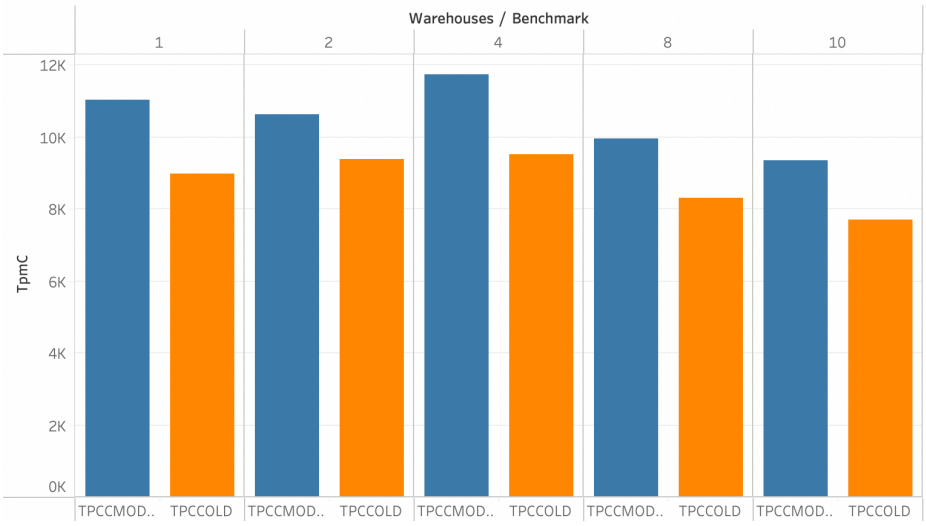


Figure 3: TPC-C PostgreSQL results

Figure 4: TPC-C MongoDB results

The original model was modified by embedding the order lines entity within an order, the exact change made in [34]. This change is because order and order lines have a composition relationship (orders are composed of order lines), so embedding is preferred.

This observation was made during the preliminary design when inspecting the relationships and directly impacted the design choice. Moreover, since the application interface requires fewer round trips to the database (no JOINS required), scalability was improved (contribution from the empirical experiments). This impact was envisioned during the preliminary design (SE) by inspecting the interface between the database and the application.

This result validates that both databases behave similarly to changes in the underlying model, even with complex transactions. Therefore, a thorough preliminary design, more specifically the allocation of requirements to data-based design as derived from application level, results in improved scalability, regardless of the technology. This is a very important finding.

4 PROPOSED SE-BASED DATABASE DESIGN METHODOLOGY

The design methodology is based on the systems engineering phases, as shown in Figure 2. The sections will focus on the database-specific design aspects while incorporating essential factors relating to how the system interfaces with the database. While the process is very similar to an ER approach at times, the focus and end result is very different.

4.1 Real-world problem/Input

It is critical to establish all actors that will manipulate and use the system (where data is concerned) either directly or indirectly. In other words, it should be noted if an action must be persisted in some way, cause a change in persisted data, or request data directly or indirectly. For each actor, all interactions must be elicited [8], [10].

The above is typically extracted during the requirements elicitation process at the system level [6], and it is vital to elicit requirements relating to the whole life cycle of the system as-is, to-be, and might-be.

4.2 Conceptual design

After requirements elicitation, a high-level/operational-level functional analysis is performed. From this analysis, it is crucial to identify the business rules relating to technical performance measures (TPMs) and high-level functions and interfaces. Detail on the frequency at which functions will be executed and any relationships and cardinalities that may pertain to data-related design on a conceptual level must be made visible and documented [6].

It is essential to point out that no database technology selection or modelling had been made up to this point. This is different from current methods that follow a middle-out approach.

4.3 Preliminary design

The preliminary design phase mostly coincides with the typical tasks associated with the ER approach, conceptual and logical modelling. However, it is still to be done agnostic of the database technology.

In the SE approach, one performs a refined functional analysis on a subsystem level. As the design progresses, the documentation has to be expanded to include new requirements or constraints that may arise from other subsystems (such as the application) or functions that interface with the database [6].

At this stage, the high-level data entities that have to be persisted, their relationships and cardinalities and some data fields or attributes should be known (from requirements and functional analysis).

Each relationship has to be defined as one of (from weakest to strongest in terms of dependence for existence) [16], [36]:

- Association - any logical relationship with no dependence on existence;
- Aggregation - parts of a whole but can exist separately;
- Composition - parts of a whole but cannot exist separately;
- Inheritance - specialization of an entity.

Given the relationships of a data entity, one can determine whether an entity is strong or weak. A strong entity's existence is not dependent on its relationships.

Cardinalities are often expressed vaguely, such as one-to-many and many-to-many, but it is more instructive to have specific descriptions such as one-to-fifty.

Closely related to cardinality is the rate at which entities (and by extension relations) are inserted into or removed from the database and updated or read. This rate can be derived from related functions and interfaces (application, for instance) and their specifications [7].

To support multi-tenancy, which enables a single system to service multiple tenants, annotate entities or relationships that may aid segregation. For example, the entity "account" could be used to segregate all other entities based on which account they belong to.

The entities and relationships can be modelled and refined in UML class diagrams with all the metadata and documentation gathered during the above steps. This process must be iterative to address all the requirements as described below.

4.3.1 Entities

Firstly, all data entities and their attributes are rendered into a UML class diagram, followed by their relationships and cardinalities.

When modelling and refining entities, the decision has to be made whether to embed or reference an entity.

As a first step, entities that grow boundless should always be strong standalone entities, and these entities will serve as anchors for other entities based on their relationships. One exception is weak entities that grow boundless, and these can use a special bucket pattern for embedding or remain a top-level entity [37].

4.3.2 Attributes

All attributes garnered from previous steps are annotated in the UML class diagrams. These should use simple types and allow for some special types, such as arrays of simple values initially. Attributes typically assigned as relational data (addresses or phone numbers, for example) should remain intact until further decomposition is required.

Imbue each entity with a unique identifier, if not already inherent. Here it is important to introduce any multi-tenancy attributes that may be used for discrimination.

4.3.3 Relationships

Any relationship is inherently bi-directional, but, in the context of a database system, some relationships can be considered single-ended or uni-directional, such as when the system only accesses the data in one direction.

Firstly, simplex relationships will be handled. These are defined as relationships between entities which do not have more than one strong relationship with other entities.

One-to-one relationships are typically embedded into the stronger of the entities, and these are also typically single-ended. Annotate a new field to denote the relation, but type it with the type of the entity embedded.

One-to-few relationships embed or reference the “few” entities into the “one” entity. The rule of thumb is based on the strength of the relation, and deciding which side to embed/refer to is based on the entities’ strength. Composition or inheritance relations should usually embed, while it is not as clear for weaker relations. It is also possible to reference the entity but embed some data that changes rarely and is frequently required.

One-to-many typically only reference each other but optionally embeds data locally that is often required and rarely changing. Typically one would refer to the “one” from the “many” side, though it is possible with a bucket pattern to have it the other way around. Use the latter approach if the directionality of operations would benefit from such an approach.

Many-to-many can be grouped into the following varieties: many-to-many, many-to-few and few-to-few.

Pure many-to-many cannot consider full embedding. Often, a new JOIN entity that references the left and right-hand entities is required.

Many-to-few can be handled similarly to a one-to-many relationship, where the “few” are embedded or referenced on the “many” side. JOIN entities are not required.

Like many-to-few, few-to-few relationships can embed or reference on both ends, depending on the association type.

Only simplex situations have been considered up to now. When more complex relationships exist where entities have different cardinalities and association types with more than one entity, one should carefully consider how to render each entity optimally. The above process should be followed recursively, rendering the stronger relationships first until only simplex relationships remain. Considering that boundless entities serve as anchors, start with these and render each association with them accordingly.

The primary decision to be made when modelling entities is whether to embed, reference or follow a hybrid approach. It is here where application query patterns play a vital role in determining the necessary trade-offs to consider.

When considering embedding in complex cases, the following questions have a bearing on the design choice [13]:

- How often is the embedded data changed?
- How often is the embedded data required in a query?

The answer to these questions will determine whether embedding is ideal. When embedded data changes often, it must be fanned out to all other entities that embed this entity on change. If the ratio to writes versus reads is skewed to the write-heavy side, referencing might prove more performant and maintain a stricter level of consistency. However, if the ratio is skewed towards the read-heavy side, embedding might prove more performant by removing the need to join the entities during a query [13]

Referencing is typically better for scaling writes for complex cases primarily by avoiding the fan-out discussed earlier. Embedding, in contrast, is better for read performance. A third option is to utilize both models, following an eventual consistency approach to embedded data. Attributes typically required in queries should be noted as possible candidates for partial embedding for this approach. A fourth option could be to utilize the bucket pattern, which creates buckets of references or embeddings.

The experiments showed that writes are more costly on a database system than reads and should be prioritised when designing complex relationships.

4.4 Detail design

Following the preliminary design, a detail design requires technology choices. The database management system (DBMS) should be chosen based on the system's constraints, controls, resources, inputs, and outputs. Form follows function; therefore, the database chosen should have a form and fit to meet the function [38].

The UML class diagrams must be synthesized into the DBMS of choice during detail design.

Indexes must be designed, which was seen as the most impactful parameter in the empirical tests. To successfully design the indexes, one has to consider the frequency at which queries will be performed. It may be more desirable to allow longer query times than to suffer lower write throughput for all inserts/updates. However, if a query is performed frequently, and especially if the collection/table grows significantly large, it cannot be helped but to add an index [29].

The rule of thumb for indexing multiple columns – start with the most selective column first, then work toward the least selective column, superseded by the clause that the left-most columns have to be fully specified to benefit from the right-most columns [9].

For scalability, one may have to consider sharding. Both databases can partition (or shard) based on a range of keys (similar to an index) or by hash. Sharding, in addition to regular partitioning, migrates a partition to a different server altogether. Not only is disk performance scaled, but also reads, writes and CPU-bound work [20], [39].

To obtain high availability (on the database layer), the database layer requires replication to serve as live backup and recovery [20], [21].

4.5 Artefact/Output

After the processes described above, the design of the database subsystem is complete. All interfaces and functions interacting with the database are defined and documented; thus, the application could be developed and implemented in parallel.

Considering the empirical results as a guide to database operations design in the application, one can make the best use of the underlying database system. For example, one can improve the system's performance by employing multiple threads and batch operations.

5 CONCLUSION

This paper has demonstrated that traditional database design methodologies lack a holistic system-wide approach. Although similarities between the phases of ER and SE exist, they are significantly different in nature, purpose, and execution. As alluded to earlier, ER follows a middle-out approach while SE follows a top-down approach.

The most challenging yet impactful decisions must be made during the preliminary design phase, which is a phase typically not included in traditional database design. In SE, a conceptual design closely represents the real world and its requirements, while preliminary designs represent the abstract database system more closely. Translation between concept requirements (Type A) and preliminary (Type B) is critical to ensure requirements are fully managed. Detail design progresses to closely represent the physical database system and, ultimately, the application’s structures. In other words, the conceptual design closely resembles the use cases and actors in the system, while the preliminary design, prudently so, more closely resembles the application and system.

Empirical database tests were performed on PostgreSQL and MongoDB to determine the databases’ characteristic performance with respect to parameter changes. From the results, literature, and systems engineering principles, a new methodology was developed in the form of a decision support framework encapsulated in the development method.

To validate that the empirical results apply to complex transactions and a modern modelling paradigm applies to PostgreSQL and MongoDB, a TPC-C benchmark was performed on traditionally modelled and modern approaches. A similar experiment was done by [34] for MongoDB. In both cases, the modern approach achieved up to 20% improvement in transactions per minute.

These results demonstrate that following a system-oriented design and modelling within a modern paradigm delivers better scalability and design flexibility, whether applied to MongoDB or PostgreSQL. For this reason, the SE preliminary design phase can and must be completed before choosing a database technology. Moreover, considering surrounding functions and interfaces in the design (such as the application), a complete solution with the correct form and fit is obtained.

6 REFERENCES

[1] P. P.-S. Chen, "The entity-relationship model---toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36, Mar. 1976, doi: 10.1145/320434.320440.

[2] A. Badia and D. Lemire, "A call to arms," *ACM SIGMOD Record*, vol. 40, no. 3, p. 61, Nov. 2011, doi: 10.1145/2070736.2070750.

[3] E. Meijer, "All your database are belong to us," *Commun ACM*, vol. 55, no. 9, p. 54, Sep. 2012, doi: 10.1145/2330667.2330684.

[4] A. A. Imam, S. Basri, R. Ahmad, and M. T. González-Aparicio, "Schema proposition model for NoSQL applications," *Advances in Intelligent Systems and Computing*, vol. 843, pp. 30–39, 2019, doi: 10.1007/978-3-319-99007-1_3.

[5] A. Kanade and A. Gopal, "A Novel Approach of Hybrid Data Model in MongoDB.," *IUP Journal of Computer Sciences*, vol. 9, no. 3, 2015.

[6] B. S. Blanchard and W. J. Fabrycky, *Systems Engineering and Analysis*, 5th ed. Harlow: Pearson Education Limited, 2014.

[7] A. A. Imam, S. Basri, R. Ahmad, J. Watada, M. T. Gonzlez-Aparicio, and M. A. Almomani, "Data modeling guidelines for NoSQL document-store databases," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 10, pp. 544–555, 2018, doi: 10.14569/IJACSA.2018.091066.

[8] C. Coronel, S. Morris, and P. Rob, *Database Systems: Design, Implementation, and Management*, 11th ed. Stamford, CT: CENGAGE Learning, 2009.

[9] B. Schwartz, P. Zaitsev, and V. Tkachenko, *High Performance MySQL: Optimization, Backups, and Replication*, 3rd ed. Sebastopol: O'Reilly Media, Inc., 2012.

[10] P. Ponniah, *Database design and development: an essential guide for IT professionals*, 1st ed. New Jersey: John Wiley & Sons, Inc., 2003.

[11] A. De Lucia, C. Gravino, R. Oliveto, and G. Tortora, "An experimental comparison of ER and UML class diagrams for data modelling," *Empirical Software Engineering*, vol. 15, no. 5, pp. 455–492, 2010, doi: 10.1007/s10664-009-9127-7.

[12] K. Chodorow and M. Dirolf, *MongoDB: the definitive guide*, 2nd ed. Sebastopol: O'Reilly, 2013.

[13] W. Zola, "6 Rules of Thumb for MongoDB Schema Design: Part 3," 2014. https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-3 (accessed Apr. 06, 2022).

[14] G. L. Sanders and S. S. S. Shin, “Denormalization effects on performance of RDBMS,” *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, vol. 00, no. c, pp. 1–9, 2001, doi: 10.1109/HICSS.2001.926306.

[15] F. Yassine and M. A. Awad, “Migrating from SQL to NOSQL Database: Practices and Analysis,” *Proceedings of the 2018 13th International Conference on Innovations in Information Technology, IIT 2018*, pp. 58–62, 2019, doi: 10.1109/INNOVATIONS.2018.8606019.

[16] P. W. Jordaan and J. E. W. Holm, “Reflection on MongoDB Database Logical and Physical Modeling,” in *2019 IEEE AFRICON*, Sep. 2019, vol. 2019-Septe, pp. 1–8. doi: 10.1109/AFRICON46755.2019.9133972.

[17] A. Kanade, A. Gopal, and S. Kanade, “A study of normalization and embedding in MongoDB,” *Souvenir of the 2014 IEEE International Advance Computing Conference, IACC 2014*, pp. 416–421, 2014, doi: 10.1109/IAdCC.2014.6779360.

[18] C. de Lima and R. dos Santos Mello, “A workload-driven logical design approach for NoSQL document databases,” in *Proceedings of the 17th International Conference on Information Integration and Web-based Applications &Services - iiWAS ’15*, 2015, pp. 1–10. doi: 10.1145/2837185.2837218.

[19] P. Atzeni, F. Bugiotti, L. Cabibbo, and R. Torlone, “Data modeling in the NoSQL world,” *Computer Standards & Interfaces*, no. October, pp. 0–1, Oct. 2016, doi: 10.1016/j.csi.2016.10.003.

[20] MongoDB, “The MongoDB 5.0 Manual,” 2021. https://docs.mongodb.com/manual/ (accessed Mar. 14, 2022).

[21] The PostgreSQL Global Development Group, “PostgreSQL 11.2 Documentation,” 2019. https://www.postgresql.org/docs/11/index.html (accessed Apr. 26, 2019).

[22] C. Fahrner and G. Vossen, “A survey of database design transformations based on the Entity-Relationship model,” *Data and Knowledge Engineering*, vol. 15, no. 3, pp. 213–250, 1995, doi: 10.1016/0169-023X(95)00006-E.

[23] S. R. Schach, *Object-Oriented & Classical Software Engineering*, 7th ed. New York: McGraw-Hill, 2007.

[24] S. W. Ambler and M. Holitza, *Agile For Dummies, IBM Limited Edition*, 1st ed. New Jersey: John Wiley & Sons, Inc., 2012.

[25] T. Kraska and B. Trushkowsky, “The New Database Architectures,” *IEEE Internet Computing*, vol. 17, no. 3, pp. 72–75, May 2013, doi: 10.1109/MIC.2013.56.

[26] A. Bala and I. Chana, “Fault Tolerance-Challenges, Techniques and Implementation in Cloud Computing,” *International Journal of Computer Science Issues(IJCSI)*, vol. 9, no. 1, pp. 288–294, 2012.

[27] P. W. Jordaan and J. E. W. Holm, “Implementation and verification of a cloud-based machine-to-machine data management system,” in *2013 Africon*, Sep. 2013, pp. 1–5. doi: 10.1109/AFRCON.2013.6757739.

[28] P. Helland, “If You Have Too Much Data, then ‘Good Enough’ Is Good Enough,” *Queue*, vol. 9, no. 5, p. 40, May 2011, doi: 10.1145/1978862.1988603.

[29] R. Copeland, *MongoDB Applied Design Patterns*, 1st ed. Sebastopol: O’Reilly, 2013.

[30] P. J. Deitel and H. M. Deitel, *C++ How to Program*, 6th ed. New Jersey: Prentice Hall, 2008.

[31] S. Jacobs, “Systems Engineering,” in *Engineering Information Security*, Hoboken, NJ, USA: John Wiley & Sons, Inc., 2011, pp. 29–58. doi: 10.1002/9780470947913.ch2.

[32] T. P. P. Council, “Tpc-C,” *TPC.org*, no. February, 2019, [Online]. Available: http://www.tpc.org/tpcc/

[33] K. B. Misra, *Handbook of Performability Engineering*. London: Springer London, 2008. doi: 10.1007/978-1-84800-131-2.

[34] A. Kamsky, “Adapting TPC-C Benchmark to measure performance of multi-document transactions in mongo DB,” *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2254–2262, 2018, doi: 10.14778/3352063.3352140.

[35] R. Osman, I. Awan, and M. E. Woodward, “Application of Queueing Network Models in the Performance Evaluation of Database Designs,” *Electronic Notes in Theoretical Computer Science*, vol. 232, no. C, pp. 101–124, 2009, doi: 10.1016/j.entcs.2009.02.053.

[36] Object Management Group, “OMG Unified Modeling Language TM (OMG UML),” 2015. http://www.omg.org/spec/UML/2.5/ (accessed Nov. 28, 2016).

[37] D. Coupal and K. W. Alger, “Building With Patterns: The Outlier Pattern,” 2019.

[38] M. H. F. Wen and V. O. K. Li, “Form Follows Function: Designing Smart Grid Communication Systems Using a Framework Approach,” *IEEE Power and Energy Magazine*, vol. 12, no. 3, pp. 37–43, 2014, doi: 10.1109/mpe.2014.2301536.

[39] pgDash, “Sharding Your Data With PostgreSQL 11,” 2019. https://pgdash.io/blog/postgres-11-sharding.html (accessed Apr. 26, 2019).

Appendix D

Source Code

All tests, experiments and benchmarks, as well as compiling and running instructions, are available from the following GitHub source code repository:

`https://bit.ly/phddss`

The following QR code will also redirect to the GitHub page: