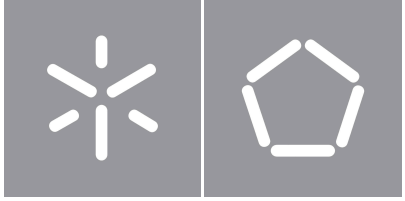


University of Minho
School of Engineering

João Ribeiro Imperadeiro

Heterogeneous database replication

october 2024



University of Minho
School of Engineering

João Ribeiro Imperadeiro

Heterogeneous database replication

Master's Dissertation in Informatics Engineering

Dissertation supervised by
Ana Luísa Parreira Nunes Alonso
José Orlando Roque Nascimento Pereira

october 2024

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:



CC BY-NC-ND

https://creativecommons.org/licenses/by-nc-nd/4.0/

Acknowledgements

This dissertation represents not only the culmination of my academic journey but also the contributions and support of many remarkable individuals.

First and foremost, I would like to express my deepest gratitude to my supervisors, Professor Ana Nunes Alonso and Professor José Orlando Pereira. Their guidance, encouragement, and expertise were essential throughout this research. Their dedication and feedback continuously pushed me to improve my work, and their mentorship has been a privilege.

I would also like to extend my heartfelt thanks to my colleagues at University of Coimbra, João Rafael Henriques and José D'Abruzzo Pereira, whose collaboration was essential to the vulnerabilities study that is part of this project. I am deeply appreciative of the generosity with which they shared their resources.

Additionally, I would like to acknowledge all the colleagues and professors that were part of my academic life, whose insights and support significantly enriched my academic experience. I am also grateful for the support and resources provided by INESC TEC.

To my family, whose love and support have been a constant source of strength: thank you for always believing in me, even in moments of doubt. Your encouragement has been my anchor, and your faith in me has been unwavering.

Finally, to my friends, who have supported and inspired me throughout this journey: thank you for your companionship, understanding, and encouragement. Your support has made even the most challenging days manageable and has been a constant reminder of the importance of balance, laughter, and resilience.

This dissertation would not have been possible without each of you. Thank you all for being a part of this journey with me.

This work is co-financed by Component 5 - Capitalization and Business Innovation, integrated in the

Resilience Dimension of the Recovery and Resilience Plan within the scope of the Recovery and Resilience Mechanism (MRR) of the European Union (EU), framed in the Next Generation EU, for the period 2021 - 2026, within project ATE, with reference 56.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. DOI 10.54499/LA/P/0063/2020

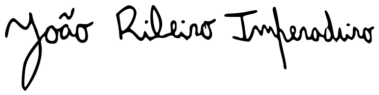
Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, october 2024



(João Ribeiro Imperadeiro)

Abstract

Replication has been a common strategy for improving the dependability of database systems. However, the use of database systems in demanding environments and critical applications calls for the enforcement of stronger dependability properties, in face of generalized fault models that include software bugs and malicious behavior. Commercial database systems assume a crash-recovery model, typically favoring a primary-backup approach for performance and scalability, which is unsuitable for Byzantine fault tolerance as a faulty primary replica can simply propagate errors to the backup replicas. Meanwhile, proposals for Byzantine fault-tolerant replication protocols can mostly be found in the literature and academic tools.

However, with very few exceptions, proposed replication protocols, even for Byzantine fault-tolerance, assume replicas are homogeneous and, at the same time, that faults are independent. We question this assumption, particularly considering the complexity of software related to databases, addressing it with a study of the diversity and known vulnerabilities in open-source database software.

Experience with existing systems, as captured by vulnerability collections, exposes the possible impact on homogeneous replication. Our study also shows there is sufficient diversity for heterogeneous replication to be feasible, allowing the use replicas with different database engines, software stacks, and even data models.

We present data that demonstrates current solutions are suboptimal and propose a novel architecture that leverages the capabilities of LLMs in code translation to enable heterogeneous Byzantine fault-tolerant replication among replicas with multiple data models. We show how LLMs can be useful with practical examples and perform an evaluation of the final system, highlighting a trade-off between fault tolerance and performance.

Keywords databases, replication, vulnerabilities, heterogeneous replication, Byzantine fault tolerance, large language models

Resumo

A replicação tem sido uma estratégia comum para melhorar a fiabilidade dos sistemas de bases de dados. No entanto, a utilização de sistemas de bases de dados em ambientes exigentes e aplicações críticas exige a aplicação de propriedades de fiabilidade mais fortes, face a modelos de falhas generalizados que incluem bugs de software e comportamento malicioso. Os sistemas de bases de dados comerciais assumem um modelo crash-recovery, favorecendo normalmente uma abordagem primária-backup para privilegiar desempenho e escalabilidade, o que é inadequado para a tolerância a falhas Bizantinas, uma vez que uma réplica primária defeituosa pode simplesmente propagar erros para as réplicas de backup. Enquanto isso, as propostas de protocolos de replicação tolerantes a falhas Bizantinas podem ser encontradas principalmente na literatura e em ferramentas académicas.

No entanto, com muito poucas exceções, os protocolos de replicação propostos, mesmo para a tolerância a falhas Bizantinas, assumem que as réplicas são homogéneas e, ao mesmo tempo, que as falhas são independentes. Questionamos esta suposição, especialmente considerando a complexidade do software relacionado com bases de dados, abordando-a com um estudo da diversidade e das vulnerabilidades conhecidas em software de bases de dados open-source.

A experiência com sistemas existentes, tal como captada pelas coleções de vulnerabilidades, expõe o possível impacto na replicação homogénea. O nosso estudo mostra também que existe diversidade suficiente para que a replicação heterogénea seja viável, permitindo a utilização réplicas com diferentes motores de bases de dados, ambientes de software, e até modelos de dados.

Apresentamos dados que demonstram que as soluções atuais são subótimas e propomos uma nova arquitetura que aproveita as capacidades dos LLMs na tradução de código para permitir a replicação heterogénea tolerante a falhas Bizantinas entre réplicas com vários modelos de dados. Mostramos como os LLMs podem ser úteis com exemplos práticos e realizamos uma avaliação do sistema final, destacando um compromisso entre tolerância a falhas e desempenho.

Palavras-chave bases de dados, replicação, vulnerabilidades, replicação heterogénea, tolerância a falhas Bizantinas, grandes modelos de linguagem

Contents

1 Introduction **1**

1.1 Problem . . . 1

1.2 Main Goals and Contributions . . . 2

2 Vulnerabilities and Diversity in Databases - Empirical Study **4**

2.1 Methodology . . . 4

2.2 Research Questions (RQs) . . . 6

2.2.1 RQ1 – What is the probability of, at any given time, a database system having a vulnerability that might be exploited? . . . 6

2.2.2 RQ2 – Does the same vulnerability impact multiple database systems at the same time? . . . 6

2.2.3 RQ3 – How has the impact and duration of vulnerabilities evolved over time? . . . 6

2.2.4 RQ4 – Is there enough variety such that a heterogeneous replicated system is feasible? . . . 6

2.3 Threats to Validity . . . 7

2.3.1 Vulnerabilities might not have been reported . . . 7

2.3.2 Considering only open source systems. . . . 7

2.3.3 Considering only DBMS vulnerabilities . . . 7

2.4 Results . . . 7

2.4.1 RQ1 – What is the probability of, at any given time, a database system having a vulnerability that might be exploited? . . . 7

2.4.2 RQ2 – Does the same vulnerability impact multiple database systems at the same time? . . . 10

2.4.3 RQ3 – How has the impact and duration of vulnerabilities evolved over time? . . . 11

2.4.4 RQ4 – Is there enough variety such that a heterogeneous replicated system is feasible? . . . 12
2.5 Related Work . . . 14
2.6 Takeaways . . . 15

3 State of the Art **17**
3.1 Heterogeneous Replication . . . 17
3.2 Fault Tolerance . . . 19
3.3 Database Replication . . . 20
3.3.1 Building Blocks . . . 21
3.4 Diversity in Databases . . . 22
3.5 Query Translation . . . 23
3.6 Takeaways . . . 24

4 Proposed Solution **25**
4.1 Proposal . . . 25
4.2 Experiments . . . 27
4.2.1 TPC-C . . . 27
4.2.2 TPC-H . . . 29
4.2.3 Cypher to SQL . . . 32
4.3 Implementation . . . 36
4.3.1 Request and Response Classes . . . 36
4.3.2 Query and Driver Interfaces . . . 37
4.3.3 Driver Implementations . . . 37
4.3.4 Server Class . . . 38
4.3.5 Client Class . . . 38
4.3.6 Query Translation . . . 38
4.4 Takeaways . . . 39

5 Evaluation **40**
5.1 Testing . . . 40
5.1.1 Basic CRUD Operations Test . . . 40
5.1.2 Incremental Update Test . . . 42
5.1.3 TPC-C Benchmark Implementation . . . 43

5.1.4 YCSB Benchmark Implementation 46
5.1.5 Test Outcomes 47
5.2 Benchmarking 48
5.2.1 Benchmark Configurations 48
5.2.2 Hardware Configurations 48
5.2.3 Results 49
5.3 Takeaways 52

6 Conclusions and Future Work **53**

List of Figures

1 Vulnerabilities timeline considering a minimum CVSS of 4.0 8
2 Number of vulnerabilities by year excluding MySQL 9
3 Number of active and reported PostgreSQL vulnerabilities by year 10
4 Average CVSS score by year 11
5 Average vulnerability fix duration (in days) by year 12
6 Number of open source databases by year (according to DB-Engines) 13
7 Number of open source databases by programming language (according to DB-Engines) 14
8 Proposed architecture using LLMs. 26

List of Tables

1 Database Engines 5
2 CVSS Ratings 5
3 TPC-C SQL queries 28
4 TPC-C SQL queries translation to PyMongo by PyTPCC and by ChatGPT 3.5 29
5 TPC-H SQL queries 30
6 Translation of TPC-H SQL queries to MQL in the baseline and by ChatGPT 3.5 31
7 Cypher queries 33
8 Cypher queries translation to SQL by baseline and by ChatGPT 34
9 TPC-C throughput for the Single-host Deployment (in req/s) 49
10 TPC-C throughput for the Multi-host Deployment (in req/s) 50
11 YCSB throughput for the Single-host Deployment (in req/s) 51
12 YCSB throughput for the Multi-host Deployment (in req/s) 51

Chapter 1

Introduction

Databases are structured collections of information, usually stored electronically, that allow for it to be efficiently stored, retrieved and managed [1]. They serve as repositories for information, facilitating the organization of data for various applications and purposes. Databases are essential components of modern information systems, supporting the storage and retrieval of data in a structured manner [2, 3, 4].

The importance of database systems goes far beyond data storage. As organizations increasingly rely on data-driven decision making, the role of databases has expanded from simple record-keeping systems to become an import component of business intelligence and real-time analytics. This transformation has been particularly evident over the past decade, with the emergence of new database paradigms and architectures designed to address the challenges of big data, cloud computing, and distributed systems.

1.1 Problem

Database systems have traditionally encapsulated the solutions to dependability challenges in data-intensive applications. First, by offering transactional recovery which ensures that the data is safe and consistent upon events ranging from lost connections to power outages and server crashes. Second, by replicating the server.

Unfortunately, database replication systems currently in widespread use assume only a benign fault model, and many, for improved performance and scalability, use a primary-backup approach. This approach implicitly assumes fault independence between replicas, an assumption that breaks down in the presence of vulnerabilities that affect all replicas simultaneously.

This is now becoming an issue as data-intensive components are sought in a growing range of systems, namely, for collecting and processing data in support of machine learning functionality, as data is increasingly considered as a valuable commodity. Furthermore, as critical infrastructure operation increasingly depends on data-supported critical information systems, dependability is increasingly required

[5]. There is also a growing concern, as these infrastructures are targeted by sophisticated attackers such as multinational criminal organizations and state-sponsored actors.

The fundamental challenge lies in achieving true fault independence in replicated database systems. If replicas are homogeneous, fault tolerance by replication is ineffective. For example, a 0-day vulnerability would affect all homogeneous replicas simultaneously, negating the benefits of replication. True fault independence requires diversity, which can be achieved through heterogeneous replication across different database implementations and, ideally, across different database models, such as relational, document-based, and graph-based.

Heterogeneous replication can provide an alternative, but has been a difficult proposition as handling the diversity of query interfaces and/or models would require a complex middleware layer, which could itself become a homogeneous point of failure.

However, there is a promising alternative that has been explored for Byzantine fault tolerant replicated systems at a larger scale and using off-the-shelf software components, by exploring diversity that already exists in the market. For example, using different operating systems [6, 7, 8] or different web servers [8].

In fact, this approach has been proposed for software components that, although large and complex, are commoditized in the sense that they can be easily replaced. For instance, most server software runs on any flavor of Linux, any other Unix-like system, and also on Windows.

However, previous proposals for heterogeneous replication of database systems were limited in the functionality of the underlying database systems that can actually be used [9, 10], mainly due to the inherent limitations of architectures based on a middleware layer and the challenges of translating between multiple query languages [11]. This happens because translating the same queries to different database systems, as needed for Byzantine fault-tolerant state machine replication, is much harder than simply replicating data among them from a primary site. The challenge is further complicated when considering replication across different database models, where not only query languages but also data models and consistency guarantees may differ significantly.

That said, the main contribution of this dissertation is a novel database system that at the same time supports heterogeneous replication across multiple database models and tolerates Byzantine faults.

1.2 Main Goals and Contributions

With the above considerations in mind, this dissertation aims to achieve the following objectives, along with specific actions to fulfill each one:

- **Assessment of Vulnerabilities in Database Systems**: Understand the extent of security challenges within current database systems, particularly regarding their vulnerability to software bugs and security flaws. For that, a thorough analysis will be conducted to evaluate the diversity among database management systems and their independence in software vulnerabilities. The study will focus on the CVE system to quantify and categorize potential risks in various database systems.

- **Comprehensive Understanding of Heterogeneous Database Replication**: Explore the state of the art in heterogeneous database replication. For that, a detailed review of the current literature on heterogeneous replication systems will be performed, examining limitations in traditional replication methods. This will also entail studying various database architectures and related fields, laying the groundwork for developing a more resilient approach.

- **Research and Implementation of a Solution**: Propose and implement a solution that replicates data across diverse databases models and tolerates Byzantine faults, for enhanced fault tolerance and security. For that, the proposed system architecture should facilitate cross-database query translations, thus supporting replication across databases with different models, such as relational, key-value, graph, and document.

- **Evaluation of the Solution**: Assess the proposed system's correctness, performance, resilience, and fault tolerance relative to homogeneous replication alternatives. For that, an extensive series of evaluations will be conducted, including functional testing and performance benchmarking using standard workloads. These evaluations should highlight both the strengths and potential trade-offs of our solution.

With these objectives, this dissertation aims to contribute significantly to the field of heterogeneous database replication, exploring the impact of vulnerabilities in database systems and advancing innovative solutions, unlocked by the use of the available database systems diversity.

Chapter 2

Vulnerabilities and Diversity in Databases - Empirical Study

Our first contribution is an evaluation of the extent to which diversity exists in database management systems and how they are truly independent in regard to software vulnerabilities, with the goal of assessing the effectiveness of current homogeneous replication approaches and building a strong case for heterogeneous replication strategies as more robust alternatives.

For that, we focused on four areas that we deemed important for a deep understanding: the probability of vulnerability exploitation in database systems at any given time, the potential for vulnerabilities to impact multiple database systems simultaneously, the evolution of vulnerability impact and duration over time, and the feasibility of heterogeneous replication given the current database ecosystem diversity.

2.1 Methodology

Our study is based on a dataset of vulnerabilities generated by a tool [12] that tracks database vulnerabilities, built over the data made available by an online CVE-tracking website [13]. At the time of writing, the tool was tracking over 14000 vulnerabilities that affected 16 projects.

The set of database engines selected for analysis is depicted in Table 1. The selection criteria was to sample diversity in database models, programming languages and level of market popularity. For each database, reported CVEs were collected and used in the study.

The analysis used CVSS (Common Vulnerability Scoring System) scores, a method used to obtain a qualitative measure of severity. However, it must not be used as a measure of risk. It consists of multiple groups of metrics, represented as a vector of variables in the form of a compressed textual representation of the values used to derive the score. Two common use cases are the attribution of severity to vulnerabilities and as a prioritization factor in the remediation of vulnerabilities. A numeric value can be derived, from

Engine	Languages
Apache Cassandra	Java
Elasticsearch	Java
MongoDB	C++
MySQL	C/C++
Neo4j	Java/Scala
PostgreSQL	C
Redis	C
SQLite	C

Table 1: Database Engines

Rating	Score
None	0.0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10

Table 2: CVSS Ratings

the textual representation, which is further categorized into five levels of severity, as presented in Table 2.

As an example, consider a specific vulnerability that affected multiple versions of PostgreSQL, *CVE-2023-2454*. This vulnerability, which was officially described as `CREATE SCHEMA ... schema_element defeats protective search_path changes`, enabled an attacker with database-level `CREATE` privileges to execute arbitrary code as a bootstrap superuser. Database owners have that privilege by default and explicit grants may extend it to other users. With a CVSS score of 7.2, it was classified as a High severity vulnerability. It affected versions 11 through 15, which demonstrates that some vulnerabilities affect not one but multiple versions of a given database engine.

Statistics related to database systems were collected from the DB-Engines website [14], which provides a comprehensive list of 480 database systems, where the oldest database on record is Adabas, dating back to 1971, and their oldest open source database is PostgreSQL, dating back to 1989.

2.2 Research Questions (RQs)

To better extract actionable information, we formulate the problem as four research questions. With each one, the goal is to explore the database systems landscape and the impact of vulnerabilities.

2.2.1 RQ1 – What is the probability of, at any given time, a database system having a vulnerability that might be exploited?

The goal is to give an idea of the probability of, at any given time, a database system being susceptible to exploitation among the products analyzed.

2.2.2 RQ2 – Does the same vulnerability impact multiple database systems at the same time?

The goal is to explore if the vulnerabilities are self-contained within a product or if the radius of exploitation goes beyond a single database product.

2.2.3 RQ3 – How has the impact and duration of vulnerabilities evolved over time?

The goal is to analyze how the severity of vulnerabilities, and the time it takes for them to be found and fixed, has evolved with time.

2.2.4 RQ4 – Is there enough variety such that a heterogeneous replicated system is feasible?

The goal is to assess whether sufficiently diverse database systems exist and how likely it is for vulnerabilities to affect multiple database versions and/or database systems. Both aspects are critical to assessing the feasibility of implementing a heterogeneous replication system, to overcome the lack of fault independence in homogeneous replication.

2.3 Threats to Validity

2.3.1 Vulnerabilities might not have been reported

Given the time it takes for vulnerabilities to be found and reported, the data for recent years might be incomplete, which might introduce a bias towards older and more widely used systems. Also, some bugs with the characteristics of a vulnerability might not be considered to be an actual vulnerability by the maintainers of the engines.

2.3.2 Considering only open source systems.

By considering only open source database server software, there is a whole category of commercial software that is excluded from the analysis. These software packages and cloud-based services might receive better support and be patched faster, but they can be out of reach for many companies.

2.3.3 Considering only DBMS vulnerabilities

Only the vulnerabilities of database server software are considered in the analysis. However, each database server relies on a software stack, which includes the operating system, shared libraries such as `libc` and programming language vulnerabilities.

For example, Log4Shell was a critical security vulnerability in the Apache Log4j logging library for Java, allowing remote code execution and affecting various applications and systems. While this vulnerability has associated CVEs, these are not listed with CVEs for databases systems written in the Java programming language, which are part of this study. Still the companies/organizations responsible for affected database systems reported this issue in advisories, recognizing its critical relevance.

2.4 Results

2.4.1 RQ1 – What is the probability of, at any given time, a database system having a vulnerability that might be exploited?

Figure 1 shows a timeline of the versions of multiple database engines, considering the CVEs that were attributed a CVSS score greater than or equal to 4.0. The version chosen for each engine was the oldest still supported at the time, given that those allow to dive into a more historical analysis. The newest

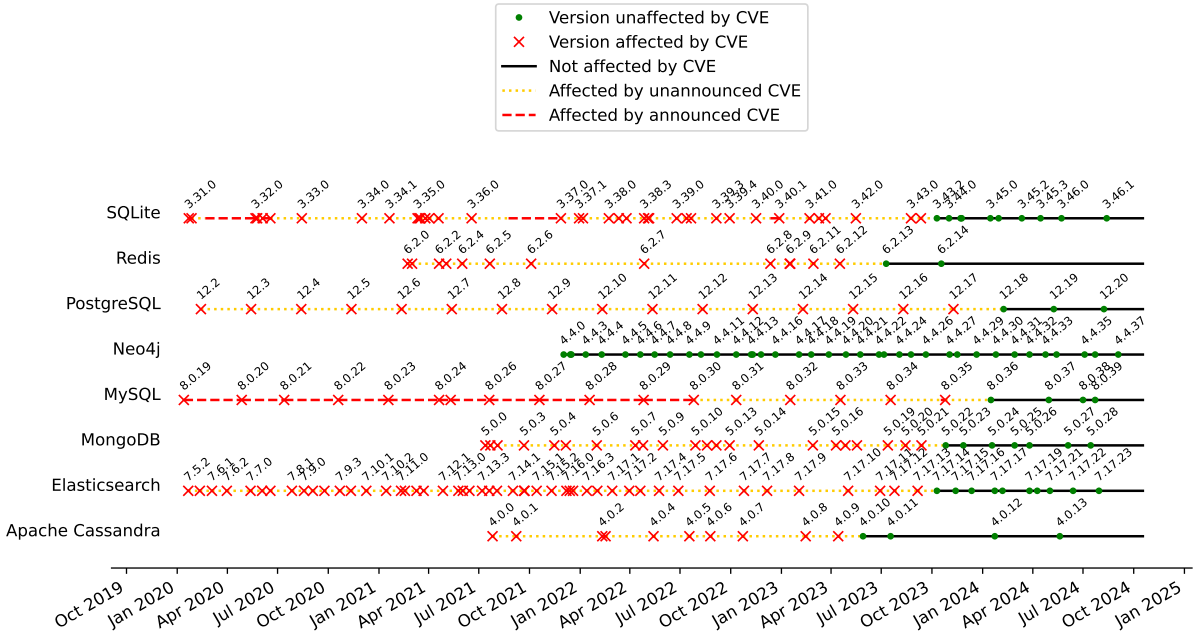


Figure 1: Vulnerabilities timeline considering a minimum CVSS of 4.0

versions are too recent and only a few vulnerabilities have been reported, which does not mean there are no active vulnerabilities, only that these have yet to be reported.

First of all, apart from Neo4j, which surprisingly is not impacted by vulnerabilities in the versions analyzed, most of the versions of the engines end up having a vulnerability reported. This does not apply to newest versions, which can be explained by the long time it usually takes for vulnerabilities to be found and reported. In short, most database engine versions are affected by at least one vulnerability and most vulnerabilities are reported only after the release of a new fixed version.

Then, there are multiple points in time, displayed as red lines, where the latest version of a database has a publicly reported vulnerability but there is no fixed version available. This means there are moments where database operators are running a vulnerable version of a database with no patch available!

While not visible in Figure 1, the dataset shows that there are multiple vulnerabilities that affect different versions of the same database engine. In fact, most CVEs are announced for multiple versions and the fixes for multiple versions are released at the same time. For example, from the previous example, vulnerability *CVE-2023-245* affects versions 11 through 15 of PostgreSQL.

Figure 2 shows the number of reported vulnerabilities by year. Data related to the MySQL database system is excluded from the figure, because there is a huge discrepancy in the number of vulnerabilities when compared with other databases. This discrepancy can be attributed to the way CVEs are associated to a product in the dataset. For example, *CVE-2019-16168* is part of the list of vulnerabilities for MySQL,

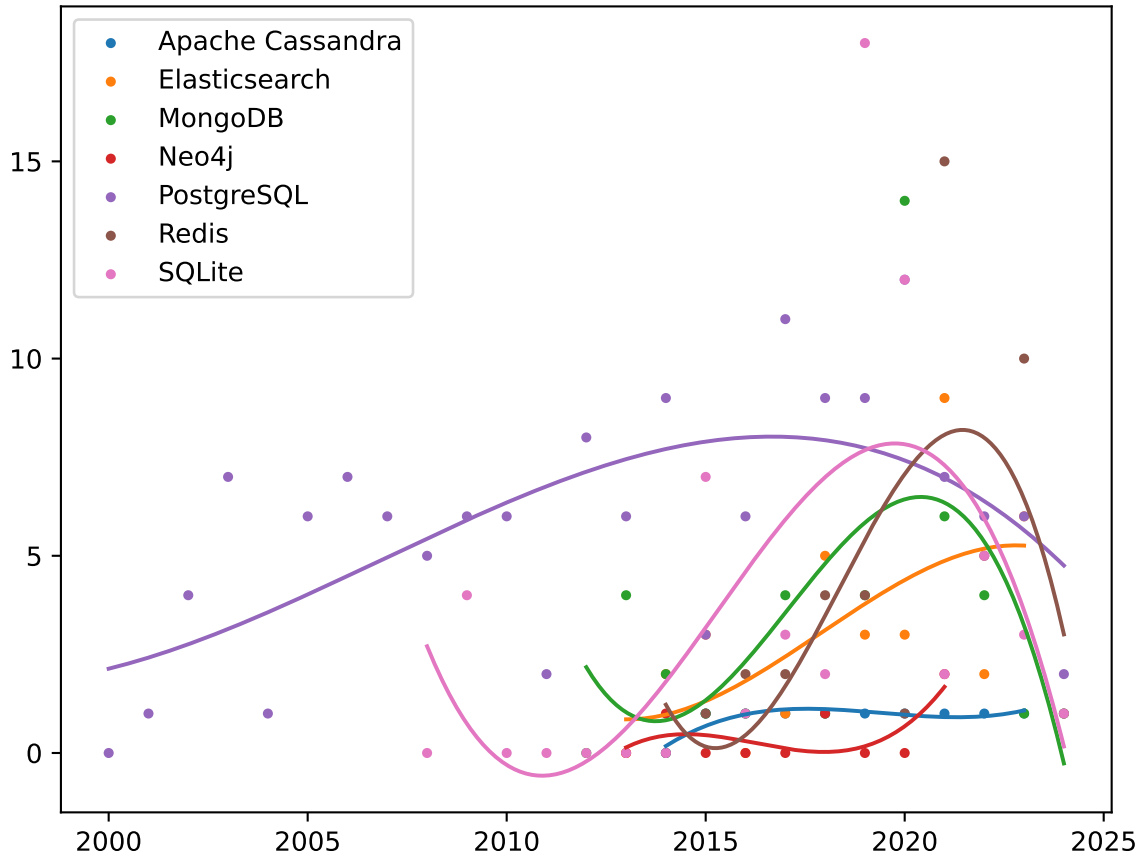


Figure 2: Number of vulnerabilities by year excluding MySQL

but it is in fact a SQLite vulnerability that affects MySQL Workbench, a visual tool unrelated to the database server. But because of the way vulnerabilities were collected from the Oracle-related products lists, this vulnerability in SQLite, which is used by MySQL Workbench, got associated with MySQL.

From the analysis, apart from more recent years, where data does not contain potentially undiscovered vulnerabilities as discussed above, there is a slight trend upwards in the number of vulnerabilities.

Figure 3 shows the relation between active and reported vulnerabilities in PostgreSQL. This database system was selected for having one of the most consistent datasets of versions information and vulnerabilities reported, while also being a popular and widely-used open source database system. The number of active vulnerabilities, *i.e.* those that have no fix available, is for the most part superior to the number of reported ones. This means that vulnerabilities are out in the wild long before they are found and fixed. The downward trend of active vulnerabilities is, again, likely an effect of the time is takes to find and report vulnerabilities. It is expected for the active vulnerabilities line to always be above the reported vulnerabilities line as new vulnerabilities are reported.

Along with this data, we considered other factors. First, because not every bug that has the charac-

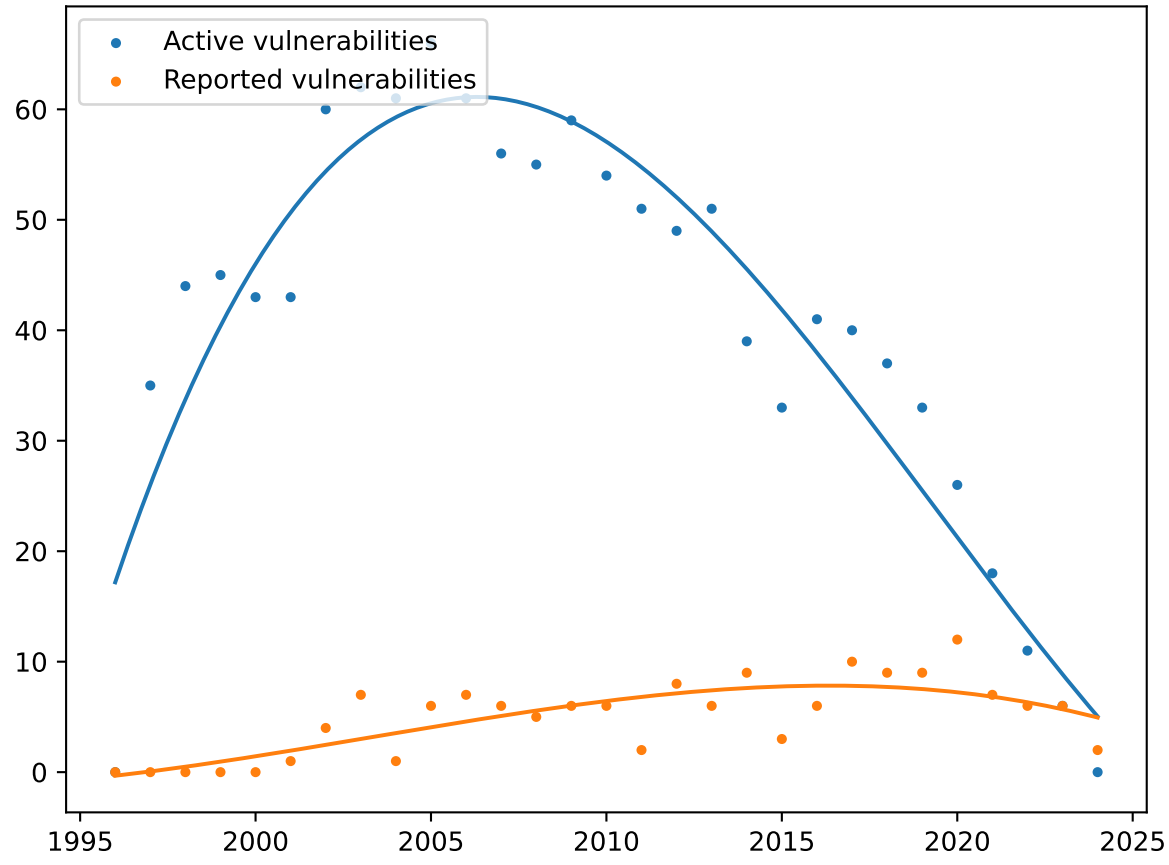


Figure 3: Number of active and reported PostgreSQL vulnerabilities by year

teristics of a vulnerability is reported as a CVE [15], the true number of vulnerabilities can be even larger. Moreover, there are issues such as Log4Shell that do not have corresponding CVEs associated, but also impact database systems indirectly. Also systems are commonly not updated as soon as patches are available [16, 17], increasing the time window during which database systems are vulnerable.

Summary: At any given time and for a given database server, there is a very high probability that there is a vulnerability affecting it.

2.4.2 RQ2 – Does the same vulnerability impact multiple database systems at the same time?

In the data we collected, no vulnerabilities affect multiple database engines, even though some use the same underlying programming language. We take this to mean that database systems are independent in terms of vulnerabilities.

There can also be vulnerabilities in the dependencies of database software or the underlying system. However, if fault independence is the goal, this can easily be overcome by carefully pairing databases that rely on different programming languages and even hardware with different characteristics.

Summary: There is no vulnerability that impacts multiple database systems.

2.4.3 RQ3 – How has the impact and duration of vulnerabilities evolved over time?

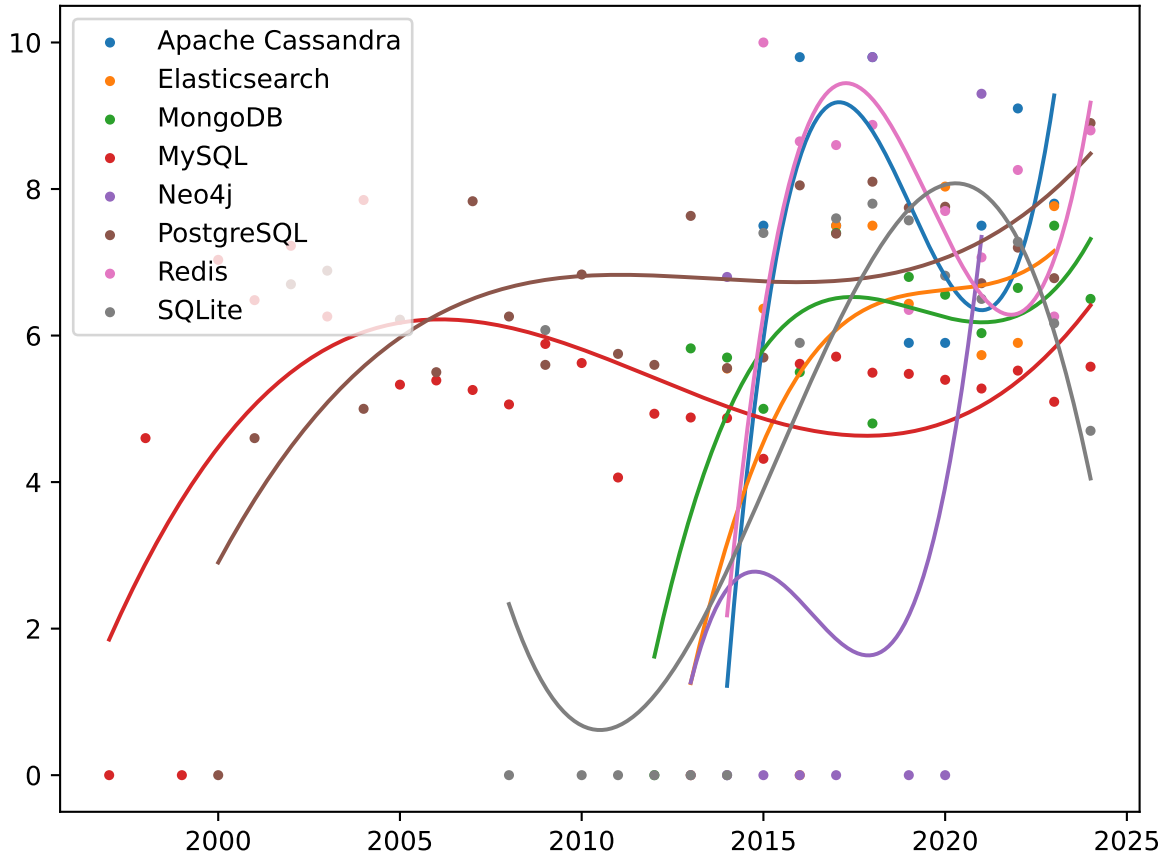


Figure 4: Average CVSS score by year

Figure 4 shows that the average CVSS score of the vulnerabilities, a measure of their severity, has been increasing and staying relatively high. This reveals a trend upwards that goes together with an increase in the number of vulnerabilities.

Figure 5 shows the number of days a vulnerability has been active from the moment the CVE is reported. The number of days that go by while a version is vulnerable is worrying, which in many cases

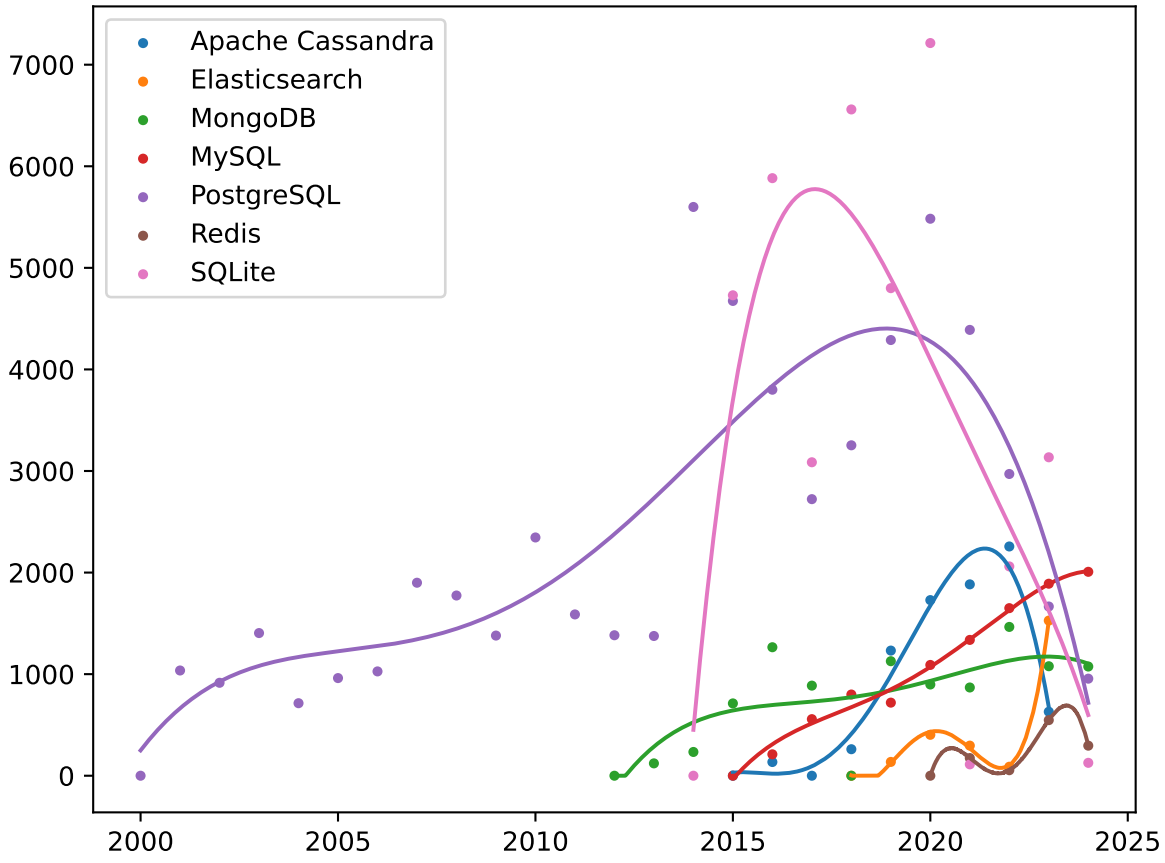


Figure 5: Average vulnerability fix duration (in days) by year

amounts to years. While depending on the database being analyzed, specifics may vary, overall we conclude that most versions of database software are vulnerable when released and it is a matter of time before vulnerabilities are found. Unfortunately, most vulnerabilities go unnoticed for years, which creates a larger time window for exploits to happen without a solution being available.

Summary: Vulnerabilities have been getting worse in terms of severity. Also, the time window between when a vulnerability is introduced and when a fixed version is released and applied is significantly large.

2.4.4 RQ4 – Is there enough variety such that a heterogeneous replicated system is feasible?

According to the analysis of the data from DB-Engines, in Figure 6, we can see that the number of open source database systems has been growing over time. Both the number of available SQL and NoSQL

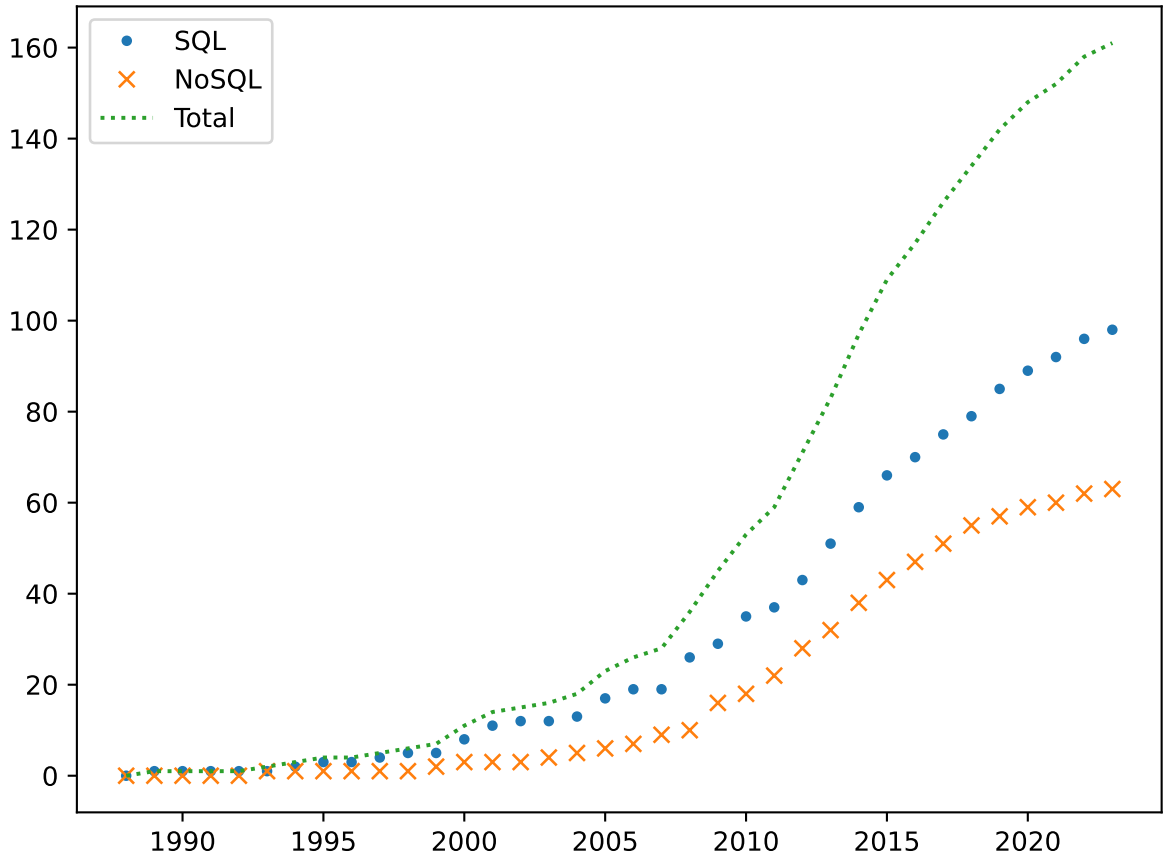


Figure 6: Number of open source databases by year (according to DB-Engines)

database systems have been steadily growing for years. Some databases were excluded from this analysis, as there is some missing information, necessary for identifying these correctly. As such the data provides a lower bound, and the real numbers are probably higher. This gives great flexibility for implementing a heterogeneous replicated database system, as there are a lot of possible databases to choose for the system.

Figure 7 shows database systems have good diversity when it comes to programming languages. Although some languages are more popular than others, there is a wide range of options to choose from. Again, this is good news regarding the heterogeneous replication approach, as one can select database systems using widely different programming languages, further reducing the possibility of overlapping vulnerabilities.

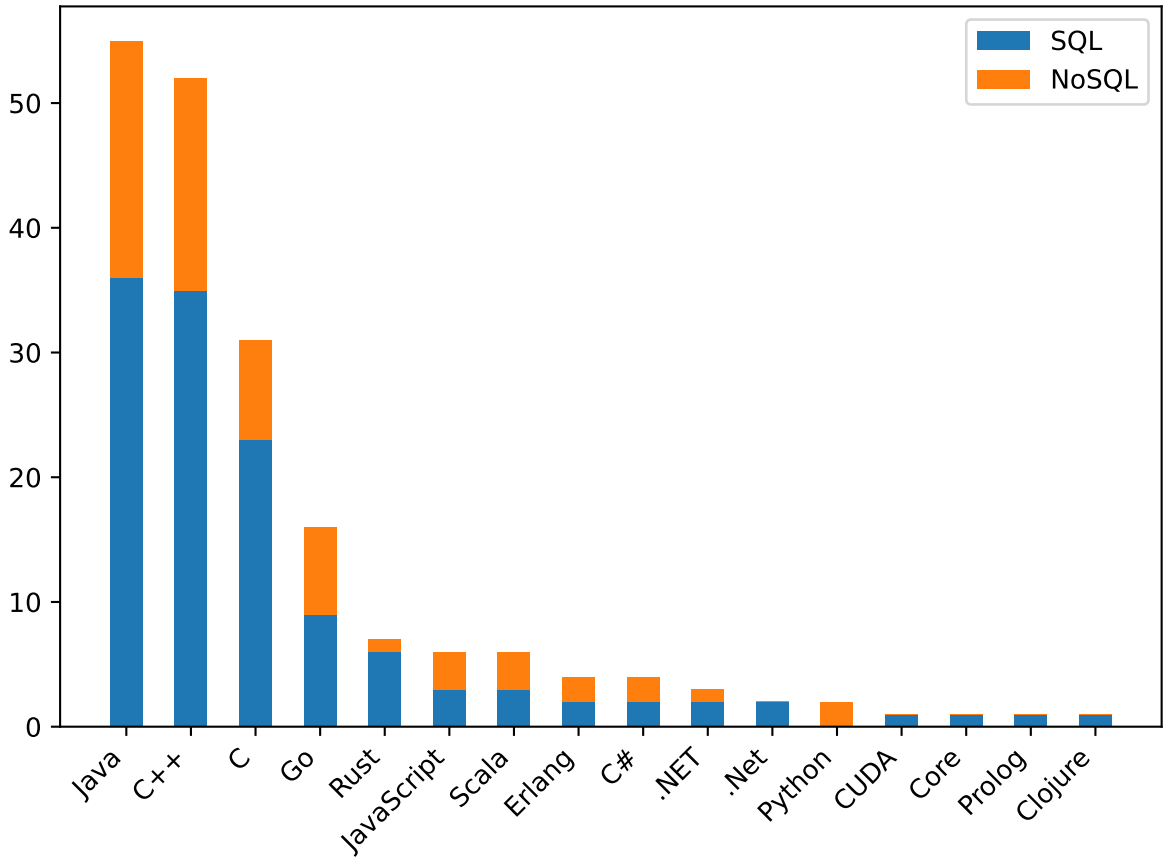


Figure 7: Number of open source databases by programming language (according to DB-Engines)

Summary: There is more than enough variety in terms of databases systems and the programming languages used, to select sets of database systems so as to make it very unlikely for vulnerabilities to affect more than one system. We conclude that our analysis supports that heterogeneous replication is feasible, and would be successful in providing fault independence to replicas.

2.5 Related Work

Previous work has analyzed the impact of software vulnerabilities on database systems. An analysis of the impact of bugs on multiple DBMSs shows that bugs that affect more than one database engine are rare [9]. Apart from bugs that result in the crash of the system, it is also concerned with more subtle bugs, including those that produce incorrect results or cause data corruption. It concludes that using different versions of the same database software provides advantages, but not as much as using different DBMSs. It also highlights that a middleware for bridging the application to such a system would have restrictions

on the subset of SQL used to interact with the data. However, this study was conducted before many of the currently available database systems were introduced.

An analysis of MySQL's bug database [15] has shown 499 to 587 misclassified bugs. This is an estimated increase of vulnerabilities between 657% and 772% over the number publicly reported in the National Vulnerability Database (NVD). As a consequence, the motivation for pursuing heterogeneous replication might be even stronger.

Considering a more general scope, beyond database server software, there are also surveys of the current state of the challenges, approaches, tools and practices related to software security patch management [16] and of the reasons for the delays in security patch application in the healthcare sector [17]. Together, they make clear that the field at large is not very responsive when it comes to patching vulnerable systems.

2.6 Takeaways

We have shown that almost all database versions end up being affected by at least one vulnerability and that most vulnerabilities affect different versions of the same database engine. This discards a potential solution that could be proposed, of using different versions of the same database engine. It is possible that this could be marginally more secure, but from the results highlighted it probably does not outweigh the disadvantages of having to manage a system consisting of multiple versions.

There appears to be an upward trend in the number of vulnerabilities being reported each year, and the number of active vulnerabilities seems to be much higher than the number of those reported. It seems likely that there are vulnerabilities that have been active in recent years and that have yet to be found and reported.

Given the analyzed vulnerability dataset, and knowing that there are many times more vulnerabilities than those reported [15], we can conclude that zero-day vulnerabilities are a real problem and must be taken into account when designing reliable systems.

As most database engine versions end up being vulnerable, database system managers should take this into account during the deployment and operation of database systems. Although it is true that vulnerabilities can be announced after the release of a new version that contains a fix, systems are not immediately updated [17], so from the announcement of a vulnerability to the deployment of the fixed version there can be a large window of time when the system is vulnerable. Hence, updating database systems must be taken into account when managing such systems.

We have seen there are no vulnerabilities in common for different database engines, which confirms the hypothesis that heterogeneous replication is a viable solution. Also, issues with the underlying system and programming language can be overcome by pairing together databases that rely on different software stacks and deploying them on machines with different hardware and software configurations.

We have shown the severity of the vulnerabilities has been getting worse and the time it takes for a vulnerability to be fixed is quite high. So, those who maintain database systems need to dedicate more resources each year to their maintenance, given the severity of the vulnerabilities, and to the assessment of potential breaches, given the span of time their databases could have been vulnerable for.

It was also clear from the data presented that there is more than enough diversity in terms of open source databases, the programming languages they use, and in the models they support. That is, if before it could be argued that there were not enough database choices to support heterogeneous replication, after this empirical study it is more than clear that it is not the case.

Summing up, with this empirical study we have shown that current homogeneous database replication systems are not satisfactory in the presence of zero-day vulnerabilities, which we have shown are an increasing threat. To avoid this and improve the fault tolerance of database systems, we should introduce diversity in the system. One way to achieve this is by implementing heterogeneous replication, which in the context of databases means using different database engines together. It is traditionally implemented by pairing databases with the same model, usually relational, but with all the available diversity we aim to overcome this restriction and unlock the pairing of databases with arbitrary models.

Chapter 3

State of the Art

Heterogeneous replication is a form of database replication in which data replicas are distributed across diverse database management systems. This approach can be valuable but poses significant challenges, such as the complexity of managing cross-system consistency, handling various failure models, and translating queries among diverse database languages.

This chapter provides an in-depth review of the state of the art in heterogeneous replication and the ongoing research into solutions that address its specific challenges. Section 3.1 examines existing implementations and the obstacles they face when handling different database models. Subsequent sections focus on methods to overcome these challenges, exploring fault tolerance mechanisms, advancements in query translation, and approaches to managing replication consistency across diverse data models.

This review ultimately frames the goal of this work: developing a heterogeneous replication solution that leverages multiple database models, achieves high fault tolerance, and employs novel query translation techniques to support robust and flexible replication across diverse systems.

3.1 Heterogeneous Replication

Heterogeneous replication can be categorized into two primary models: single-model and multi-model replication.

Single-model replication refers to the scenario where data is replicated across different instances of the same type of database system (e.g., multiple relational databases). This simplifies data synchronization and consistency management, since the underlying systems share a similar data model and language. However, it does not take full advantage of the diverse capabilities offered by different database systems.

Multi-model replication, on the other hand, enables replication across different types of database systems (e.g., relational, document, key-value, graph). This model enhances flexibility and allows organizations to utilize specialized databases for different types of read-only workloads. However, it introduces

significant complexity in managing cross-system consistency, data transformation, and handling various failure models.

A form of heterogeneous database replication is widely available in multiple products, for example, Oracle GoldenGate [18], Striim [19], Continuent TungstenReplicator [20], Jumpmind SymmetricDS [21], and Tapdata [22]. While they support multiple database models, they do not tolerate Byzantine faults (See Section 3.2 for details). In common configurations, updates are performed on a single replica and then propagated in an eventually consistent manner. This can be done statically, on an existing database, or dynamically, by capturing and propagating updates. Heterogeneity is dealt with by connectors which are capable of converting from and to each specific database system. One drawback is that not all these products are open source, which limits the knowledge we can obtain from them.

To avoid the need for different connectors, it has been proposed for the workload to be specified in terms of templates that can then be instantiated for different targets [23]. It describes a system for replicating data from a primary database to any number of replicas by introducing a separate process that stores the transaction logs for updates made to the primary. Then, there is a component that translates the commands to a syntax supported by the target databases, but this translation layer must be implemented for every new database that needs to be supported. In terms of source databases supported, the requirement is for them to provide a way to access the transaction logs. This makes it a heterogeneous replicated system that only supports the relational model and does not tolerate Byzantine faults.

DivDB [24] describes a concrete implementation of a heterogeneous replication system that tolerates Byzantine faults, composed of a middleware for interacting with heterogeneous relational DBMS, with the caveat that they must be compatible with the MySQL syntax. It works by using the BFT-SMaRt protocol and having a fixed leader. The leader applies the transactions and then uses the 2PC protocol to propagate changes to the followers.

However, this is not useful for Byzantine fault tolerance, as a faulty primary replica, in which the update is initially processed, will propagate errors to all others.

An alternative is to interpose a software layer between clients that issue database operations and database servers that execute them. This middleware can directly issue commands to all replicas, avoiding the dependence on any single one, and collect and validate replies before returning them to clients. This architecture is proposed for diverse database replication [9], but is restricted to the relational model, more specifically to a standard SQL dialect to avoid the problem of translation.

HRDB [25] follows a similar approach and focuses on improving concurrency in update processing, while being tolerant to Byzantine faults.

This approach requires that commands be translated into each target dialect, which restricts the commands that can be used and makes the middleware layer complex [11]. In fact, the complexity of this layer is itself a problem, since it might have its own vulnerabilities.

ScalarDL [10] also uses a similar architecture but bypasses the translation issue by restricting the workload to a very limited set of insert/update/delete operations of a key-value store, regardless of the underlying database system.

Thus, there is currently no heterogeneous replication solution that at the same time supports multiple database models and tolerates Byzantine faults. Furthermore, most of the systems discussed have the drawbacks of requiring a trusted component, usually a primary replica, and being eventually consistent, both of which lower the fault tolerance of the database system and leave it susceptible to data loss.

3.2 Fault Tolerance

In dependable computing, events such as lost connections and server crashes are collectively known as benign faults, where a component omits some expected behavior, and are dealt with by introducing redundancy in the sense that the missing behaviors can be repeated until successful or provided by additional components [26]. In database systems, examples of these general strategies are found in recovering from logs or switching client connections to a replica of the server. These strategies are justified by the fact that such faults are statistically independent events. For instance, server replicas are expected to have separate network and power connections, as otherwise a single fault would impact them all.

Dependable computing also deals with a second class of faults where components are expected to deviate arbitrarily from their specification, and thus can assertively output wrong results. This abstracts events such as bit-flips due to hardware malfunction, but also data corruption as a consequence of software bugs. Such faults can also be addressed with additional redundancy, to detect and then mask the offending behavior. For instance, by voting and relying on a sufficient majority while never assuming that there is a trusted component that does the tallying. This is known as Byzantine fault tolerance [27], and has seen a great deal of progress in recent years, mainly driven by research in blockchains, meaning that we have a wide variety of protocols for different trade-offs and assumptions on the environment [28, 29, 30, 31, 32].

The remaining challenge is, however, when faults are not statistically independent. For instance, a server crash caused by a software bug triggered deterministically by some input would cause all replicas to stop at the same time, thus rendering replication useless. Moreover, arbitrary behavior can also result from malicious actors controlling some components of the system. In this case, the vulnerability that

makes this possible will exist in every replica and will allow similar exploits in all of them.

A known solution is to formally prove the correctness of key software components, such that arbitrary faults result only from statistically independent events such as bit flips. A second alternative is to independently develop multiple versions of the same software component such that they do not contain the same bugs in what is called n-version or multiversion programming [33]. However, these approaches are costly and feasible only on a small scale. Given the complexity of general purpose database management systems, neither of them is viable.

Byzantium and SteelDB [34, 35] implement Byzantine fault-tolerant database replication, but do not account for diversity as they only support homogeneous replication.

A combination of Byzantine fault-tolerant replication with sharding and adaptive placement of data is proposed [36], but is restricted to a simple key-value interface.

3.3 Database Replication

Database replication is the process of distributing copies of a database across various locations [37]. It improves data availability and fault tolerance. It can be categorized into two main groups to meet different needs.

In a primary-secondary replication model, the primary server is responsible for handling write operations, while the replica servers replicate these changes to maintain a consistent copy of the data.

In contrast, multi-primary replication allows multiple servers to act as both primary and replica, enabling bidirectional data synchronization. It is useful in distributed environments where changes can occur on different servers simultaneously.

Both these models provide several benefits, such as improved fault tolerance, load balancing and scalability. In the event of a server failure, replicas can take over to ensure continuous access to the data. Additionally, distributing read operations across multiple servers can enhance performance by reducing the load on the primary server.

There is also another distinction in the way replication can be performed and which affects the consistency of data. It is related to the synchronicity of the data updates.

In synchronous replication, changes made to the primary database are immediately replicated to the secondary databases. This means that every transaction on the primary must wait for confirmation from the other replicas that the changes have been successfully applied before the transaction can be considered complete. The advantage is that it ensures all replicas are always consistent with the primary,

offering high data integrity. However, it can introduce latency in transaction processing since the primary has to wait for the replicas to confirm the data changes, which can slow down performance, especially over long distances or in cases of network issues.

In contrast, asynchronous replication does not require the primary database to wait for confirmation from the replicas before committing a transaction. Once a transaction is committed on the primary, the changes are sent to the replicas, but the primary does not wait for an acknowledgment that they have been applied. This offers much lower latency and higher throughput, making it more performant for distributed or high-volume systems. However, asynchronous replication introduces a risk of data inconsistency between the primary and replicas in the event of a failure, as replicas may lag behind the primary.

However, implementing and managing database replication comes with challenges. Database administrators must carefully configure and monitor the replication process to avoid issues such as data inconsistencies, conflicts and latency. Security considerations, such as ensuring data integrity and protecting sensitive information during replication, also play a fundamental role in designing and maintaining a robust system.

3.3.1 Building Blocks

Several established technologies can have an important role as building blocks for implementing database replication systems.

Change Data Capture (CDC) [38] represents a fundamental paradigm in database replication, enabling the propagation of data modifications from source to destination databases. While some database engines provide native CDC support, its availability varies across different database management systems, creating potential constraints for replication solutions. CDC implementations typically integrate with distributed logs or message brokers, which serve as middleware layers facilitating communication and data transformation between heterogeneous database systems. This pattern has become prevalent in many replication solutions, though the dependence on native CDC support can limit the range of supported database platforms.

Foreign Data Wrappers (FDW) [39] offer an alternative approach, specifically within the PostgreSQL ecosystem, by providing a standardized interface for accessing remote data sources. This technology enables PostgreSQL instances to interact with diverse external databases through a unified abstraction layer. However, this approach introduces limitations, notably the requirement for a PostgreSQL instance to serve as an intermediary, potentially creating a single point of failure in the system architecture.

Recent advances in database replication architectures, such as those demonstrated in CloudDBAp-

pliance [40] and Loom [41], provide valuable ideas for the implementation of a robust heterogeneous replication system. They reveal the potential for a more resilient architecture by leveraging distributed logs and middleware-based replication, strengthening fault tolerance and consistency across diverse database engines.

Both systems implement a primary-backup replication architecture mediated by a distributed log. These systems offer flexible read consistency by optionally allowing reads to bypass the distributed log and their failover mechanism relies on the log, ensuring consistency by allowing the new leader to process pending updates before resuming normal operation. This design emphasizes not only fault tolerance but also supports enhanced flexibility in read consistency, making it suitable for heterogeneous replication environments, where the diversity of the available systems can unlock novel query patterns.

Complementing these replication-focused approaches, a mechanism for Byzantine fault detection in transactional database systems has been proposed [10]. It is particularly relevant for heterogeneous database environments, as it enables proactive detection and abortion of transactions that might produce inconsistent results across different database management systems.

3.4 Diversity in Databases

In recent years, there has been a profound change in the market for database systems driven by cloud computing and big data. Instead of a few options for SQL database servers, there are now a large number of alternative proposals with different characteristics [42]. The revolution that started with NoSQL and cloud systems, and continues with NewSQL systems, has greatly improved the diversity available.

There are now NoSQL databases that support different data models such as: document-based [43]; key-value [44]; column stores [45]; and graph-based [46]. However, this makes it harder to explore such diversity, given the different query models and interfaces.

This has also led to research on heterogeneous data stores, which allow queries to be executed using the most efficient model, resulting in multistores, which enable heterogeneous data stores to be queried through a single query language, and polystores, which also support multiple query languages [47].

BigDAWG [48], while capable of migrating or replicating data between storage engines, cannot replicate commands. Thus, it is unsuitable for fault tolerance purposes, as writes on replicas cannot be considered to be independent.

An approach using polyglot engines, such as CloudMdsQL [49], to translate queries across databases with different models, while unexplored in the context of heterogeneous replication, would again lead to a

middleware-based architecture, with its lack of diversity impacting the usefulness for fault tolerance.

3.5 Query Translation

In order to unlock the use of multiple database models together, there is the need for a mechanism that helps in the conversion of queries between different database query languages. A way to achieve this is by leveraging automated query generation.

A recent technological advancement in this area are LLMs, pushing the boundaries of machine understanding and generation of language [50]. These models, typically based on transformer architectures, have reached unprecedented scales, with billions to trillions of parameters. OpenAI's GPT-4, for instance, exemplifies this trend, incorporating over 100 billion parameters, enabling it to perform diverse linguistic tasks such as translation, summarization, and even creative writing with remarkable proficiency [51].

The foundation of modern LLMs lies in the self-attention mechanism [52], which allows models to capture long-range dependencies in text. This architecture has since been scaled up dramatically, as demonstrated by models like BERT [53] and GPT [54]. While BERT focuses on bidirectional context understanding, enabling strong performance on tasks like question answering and text classification, GPT-series models emphasize autoregressive text generation, excelling in tasks requiring fluent and coherent output.

One of the key breakthroughs enabling the development of LLMs has been the use of unsupervised pre-training on vast corpora, followed by task-specific fine-tuning. This pre-training and fine-tuning paradigm allows models to learn general linguistic features during pre-training and then adapt to specific tasks, achieving state-of-the-art performance across a wide range of NLP benchmarks.

However, the scale of LLMs introduces challenges. Training such models demands immense computational resources and energy, raising concerns about environmental impact [55]. Moreover, larger models tend to exhibit problems like hallucination, where they generate plausible-sounding but factually incorrect information. Addressing these limitations while maintaining the impressive performance of LLMs remains a critical area of ongoing research.

Furthermore, the issue of leveraging LLMs to translate queries among different query languages, namely, for different data models, has not been specifically addressed in the literature. In spite of this, there has been some promising research on the translation between SQL dialects [56].

However, there are several proposals that have targeted text-to-SQL conversion, namely serving as a novel user interface for databases [57, 58, 59, 60, 61, 62, 63, 64, 65, 66].

To assess the viability of the approach, some benchmarks have been introduced [57, 67]. Data contamination has been identified as a potential issue for text-to-SQL conversion [68].

Previous attempts, without resorting to LLMs [69, 70], were limited to SQL as output.

SQL-to-text translation has also been addressed with [71] and without LLMs [72], namely, for query explainability.

Code generation has been a well-known application for LLMs [73] with tools like GitHub's Copilot, namely to generate code [74] that processes SQL queries [75]. In addition, some propose to adapt, through training, open source models for specific contexts [76].

Query translation should be suitable as a code translation task as the context can be reduced to target tables/data structures [77], even for query languages with limited available corpora [78].

Finally, while some limitations have been found in LLM-assisted code translation, a possible mitigation has also been proposed [79].

3.6 Takeaways

While there is existing research on the topics explored, the literature on heterogeneous replication may not be sufficient for definitive conclusions, signaling the need for further analysis and tool development in this area, with the development of a novel resilient heterogeneous replication system. The insights from previous solutions and database systems architectures give a solid starting point for the proposal. Also, LLMs might prove useful in overcoming the shortcomings of existing solutions.

Also, an important point is that there is currently no solution that supports databases with different models without major restrictions, as most are geared towards relational databases. Addressing this issue will be one of the goals of this dissertation, unlocking support for the major database models in a single heterogeneous database system.

Chapter 4

Proposed Solution

We propose and develop a novel architecture designed to facilitate replication across diverse database systems. This approach leverages LLMs during the application development process, effectively bridging the gap between different data models and query structures, thus eliminating the need for complex middleware solutions for query translation.

The viability of this architecture is demonstrated through a series of experiments, which provide evidence that the integration of LLMs is both feasible and practical for this purpose. A comprehensive explanation of the implementation is provided, detailing the steps involved and the decisions made during the development process.

The section concludes with a discussion that highlights the key takeaways from the research, reflecting on the implications of the results.

4.1 Proposal

In contrast to previous proposals for heterogeneous Byzantine fault-tolerant replication of database systems that are limited by the need to translate queries [11], restricting the available diversity [9] or the provided query interface [10, 36], we take advantage of current state-of-the-art Large Language Models (LLMs) to achieve both high diversity and high expressiveness. In a nutshell, we leverage LLMs to provide translation and conversion of queries between replica-supported languages, even if among different models.

Figure 8 provides a vision of a two-stage architecture for database dependability. The goal is to implement a replicated state machine [80] that not only orders requests, but also handles non-determinism in responses due to the diversity in database replicas.

In the Development stage, an application developer uses multiple LLMs to translate user-provided queries to the querying languages supported by diverse database replicas. A testing phase is required to assess the translated queries' equivalence, comparing among the results returned by the databases and

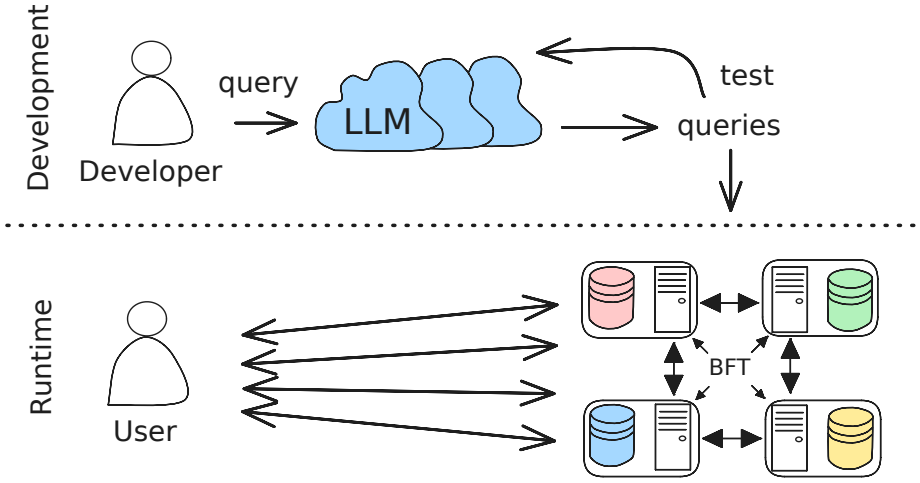


Figure 8: Proposed architecture using LLMs.

to what would be expected, if possible. Once satisfied, the developer can install the queries on the nodes of the BFT replication layer.

At Runtime, users issue queries to the BFT replication layer. First, BFT nodes agree on a global order for the execution of the request, using a formally proven Byzantine fault-tolerant protocol, for example BFT-SMaRt [29]. Then, each node executes the request in its corresponding database system. The responses of all replicas are forwarded to the user. Unlike traditional replicated state machines, responses will likely differ in format and result order. To avoid this, the results are first converted to a common relational format [81] and the queries are required to include ordering clauses. Also, the use of constraints that are only present in some databases might be a problem. For example, foreign keys can cause an abortion of certain queries in relational databases, while in other databases the same changes would be accepted, which might be a strong argument for not using them. The same applies to non-deterministic queries, which should not be used as the database servers execute the queries independently. Finally, the user selects the majority of equal results as a response to the query. If dissenting results are detected, the BFT layer can be signaled to initiate a recovery process of inconsistent replicas as in existing systems [10].

A significant advantage is that we avoid having a complex middleware layer for runtime translation of queries. In fact, complexity would defeat our proposal in two ways. First, complexity means that there is an increased likelihood of containing bugs that would be shared by all replicas. Second, the overhead of such a layer would make it harder to embed in the application itself and would require a physically separate server layer, with the corresponding administrative complications.

4.2 Experiments

To assess the viability of our approach, we conducted some experiments using LLMs.

We started by translating SQL queries to MongoDB's query language (MQL) and comparing the results with an established ground truth. We selected a set of queries from the TPC-C and TPC-H benchmarks and used OpenAI's ChatGPT 3.5 [82] (simply referred to as ChatGPT, from now on) and Google's Gemini [83] as the LLMs for our experiments. As ground truth, for TPC-C queries, we used a reference MongoDB implementation [84], and for TPC-H queries, we used another implementation [85]. Both LLMs returned correct MQL implementations for all tested queries, with results from ChatGPT most similar to our ground truth. Gemini's results, while also arriving at correct translations, tended to be more verbose, so we focus on the results for the experiments using ChatGPT.

Then, we converted queries in Neo4j's query language (Cypher) to SQL, in order to assess the generality of LLMs. In practice, it is probably more common to start from SQL syntax and needing to convert to other query languages, but this way we can assess if the reverse also works. These queries and the respective translations were collected from Neo4j's blog [86] and documentation [87].

An important caveat of this experiment (and LLMs in general) is that results are not deterministic, which means that the same prompt can yield slightly different results.

For each test, to determine the equivalence of the queries, we compared the results of executing each query on the same data.

4.2.1 TPC-C

The first set of tests use the queries provided in the Python TPC-C codebase and include `INSERT`, `SELECT` and `UPDATE` operations. This provided us with queries in SQL, (Table 3), part of the SqliteDriver, and the counterpart in the syntax of PyMongo, a Python package for interacting with the MongoDB database, part of the MongodbDriver, which can be directly compared with the LLM results. After some iterations, we found that prompts of the form

```
Only output code. Convert the following SQL query to a PyMongo query:
<SQL query>
```

with <SQL query> replaced with the target query, provided the best results, i.e., the most similar to the ground truth.

The translations of each query in Table 3, are presented in the corresponding row in Table 4.

From the result for the `INSERT` query, in Table 4, we can see there are some differences between

No.	SQL
1	`INSERT INTO NEW_ORDER (NO_O_ID, NO_D_ID, NO_W_ID) VALUES (?, ?, ?)`
2	`SELECT O_ID, O_CARRIER_ID, O_ENTRY_D FROM ORDERS WHERE O_W_ID = ? AND O_D_ID = ? AND O_C_ID = ? ORDER BY O_ID DESC LIMIT 1`
3	`UPDATE DISTRICT SET D_YTD = D_YTD + ? WHERE D_W_ID = ? AND D_ID = ?`

Table 3: TPC-C SQL queries

the PyTPCC query and the LLM-generated version. First, `insert_one` is used instead of `insert`, which is actually more correct since `insert` has been deprecated. However, in the generated query, the collection's name will need to be converted to lowercase. This could be improved by providing the schema as context in the prompt.

For the `SELECT` query (second row in Table 3), PyMongo and LLM-generated queries (second row in Table 4) are equivalent, even if the arguments to `sort` are different, as either syntax is supported by PyMongo.

For the `UPDATE` statement (last row in Table 3), there are a few differences between PyMongo and LLM-generated queries (last row in Table 4). First, `update_one` is used instead of `update`, which is actually more correct as `update` has been deprecated. The filter is also different, as the PyTPCC driver exploits the fact that it has access to the attribute `_id` when executing the query, which ChatGPT does not know. Both versions are correct, nonetheless.

We conclude that our experiments yielded promising results, demonstrating the effectiveness of LLMs in accurately translating queries, even without providing the schema to the LLMs.

Still, the output from the LLMs should not be blindly trusted as correct. This means, for our approach, that the focus should shift from query translation to query testing, which should already be part of a developer's workflow.

It could be argued that these results are biased, given the popularity of the TPC-C benchmark, which means it is probably well represented in the LLMs training dataset. We performed additional tests with fictitious versions of the queries in Table 3, keeping the overall structure, but changing the table and

No.	PyMongo in PyTPCC	SQL to PyMongo translation by ChatGPT 3.5

1	`self.new_order.insert(`
	`{"NO_O_ID": d_next_o_id,`
	`"NO_D_ID": d_id,`
	`"NO_W_ID": w_id})`
		`db.NEW_ORDER.insert_one(`
		`{"NO_O_ID": value1,`
		`"NO_D_ID": value2,`
		`"NO_W_ID": value3})`

2	`self.orders.find(`
	`{"O_W_ID": w_id,`
	`"O_D_ID": d_id,`
	`"O_C_ID": c_id},`
	`{"O_ID": 1,`
	`"O_CARRIER_ID": 1,`
	`"O_ENTRY_D": 1})`
	`.sort("O_ID",`
	`direction=pymongo.DESCENDING)`
	`.limit(1)`
		`db.orders.find(`
		`{"O_W_ID": your_value,`
		`"O_D_ID": your_value,`
		`"O_C_ID": your_value},`
		`{"O_ID": 1,`
		`"O_CARRIER_ID": 1,`
		`"O_ENTRY_D": 1})`
		`.sort([("O_ID", -1)])`
		`.limit(1)`

3	`self.district.update(`
	`{"_id": d["_id"]},`
	`{"$inc": {"D_YTD": h_amount}})`
		`db.DISTRICT.update_one(`
		`{"D_W_ID": your_value,`
		`"D_ID": your_value},`
		`{"$inc": {"D_YTD": your_value}})`

Table 4: TPC-C SQL queries translation to PyMongo by PyTPCC and by ChatGPT 3.5

column names to random strings (AAA, BBB, CCC, etc.).

The results were equivalent to the original TPC-C in Table 3, with the changes having no functional effect in the translation.

4.2.2 TPC-H

The second set of tests evaluates the performance of the translation for significantly more complex queries, from the TPC-H benchmark. We selected TPC-H queries Q1 and Q4 from a codebase that provides the queries in SQL (Table 5) and counterparts in MQL (Table 6).

We found that prompts of the form

```
Only output code. Given the schema, convert the following SQL query to a MongoDB query:
<collection1> collection:
<collection1 document example>

<collection2> collection:
<collection2 document example>

...

Query:
<SQL query>
```

No.	SQL

Q1	select
	  l_returnflag, l_linestatus, sum(l_quantity) as sum_qty,
	  sum(l_extendedprice) as sum_base_price,
	  sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
	  sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
	  avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price,
	  avg(l_discount) as avg_disc, count(*) as count_order
	from lineitem
	where l_shipdate <= date '1998-12-01' - interval '[DELTA]' day (3)
	group by l_returnflag, l_linestatus
	order by l_returnflag, l_linestatus

Q4	select o_orderpriority, count(*) as order_count
	from orders
	where
	  o_orderdate >= date '[DATE]'
	  and o_orderdate < date '[DATE]' + interval '3' month
	  and exists (select * from lineitem
	  where l_orderkey = o_orderkey
	  and l_commitdate < l_receiptdate)
	group by o_orderpriority
	order by o_orderpriority

Table 5: TPC-H SQL queries

yielded the best results, where <collectionX> should be replaced by the collection's name and <collectionX document example> with an example document for the collection, for each relevant collection, and replacing <SQL query> with the desired query. In contrast to the tests based on TPC-C (Section 4.2.1), it was necessary to include the schema for the collections as context in the prompt, as otherwise the results of the translation were frequently incorrect.

Regarding Q1, in the first row of Table 5 and Table 6, the translation was mostly successful on the first try, only requiring changing the date in the first `$match` to the one desired. When compared to the baseline, manual implementation, there were some relevant differences, such as the order of the pipeline to compute the aggregate. On a side note, ChatGPT's translation includes additional fields (e.g. `avg_qty`), missing from the manual translation, but specified in the SQL query.

Regarding Q4, in the second row of Table 5 and Table 6, required an iteration with the LLM. When prompting for the first time, the resulting query had a `$match` in the aggregate pipeline that was syntactically valid but incorrect in regard to the expected result:

No.	MQL baseline	Translation by ChatGPT 3.5

Q1	db.lineitem.aggregate([
	{$match:{
	SHIPDATE:{
	$lte: ISODate("2016-01-01T00:00:00.000Z")
	}
	}
	},
	{
	$project:{
	RETURNFLAG:1,LINESTATUS:1,QUANTITY:1,
	EXTENDEDPRICE:1,DISCOUNT:1,
	l_dis_min_1:{ $subtract:[1, "$DISCOUNT"] },
	l_tax_plus_1:{ $add:["$TAX", 1] }
	}
	},
	{
	$group:{
	_id:{RETURNFLAG:"$RETURNFLAG",
	LINESTATUS:"$LINESTATUS"},
	sum_qty:{$sum:"$QUANTITY"},
	sum_base_price:{$sum:"$EXTENDEDPRICE"},
	sum_disc_price:{
	$sum:{$multiply:["$EXTENDEDPRICE",
	"$l_dis_min_1"]}
	},
	sum_charge:{
	$sum:{
	$multiply:[
	"$EXTENDEDPRICE",
	{$multiply:["$l_tax_plus_1", "$l_dis_min_1"]}
	]
	}
	},
	avg_price:{$avg:"$EXTENDEDPRICE"},
	avg_disc:{$avg:"$DISCOUNT"},
	count_order:{$sum:1}
	}
	},
	{$sort:{RETURNFLAG:1, LINESTATUS:1}}
	])
		db.lineitem.aggregate([
		{
		$match: {
		SHIPDATE: { $lte: ISODate("1998-11-28T00:00:00.000Z") }
		}
		},
		{
		$group: {
		_id:{RETURNFLAG: "$RETURNFLAG",LINESTATUS: "$LINESTATUS"},
		sum_qty: { $sum: "$QUANTITY" },
		sum_base_price: { $sum: "$EXTENDEDPRICE" },
		sum_disc_price: { $sum: { $multiply:
		["$EXTENDEDPRICE", { $subtract: [1, "$DISCOUNT"] }]}},
		sum_charge: { $sum: { $multiply: ["$EXTENDEDPRICE", { $multiply:
		[{ $subtract: [1, "$DISCOUNT"] }, { $add: [1, "$TAX"] }]}]},
		avg_qty: { $avg: "$QUANTITY" },
		avg_price: { $avg: "$EXTENDEDPRICE" },
		avg_disc: { $avg: "$DISCOUNT" },
		count_order: { $sum: 1 }
		}
		},
		{
		$project: {
		_id: 0, l_returnflag: "$_id.RETURNFLAG",
		l_linestatus: "$_id.LINESTATUS", sum_qty: 1,
		sum_base_price: 1, sum_disc_price: 1,
		sum_charge: 1, avg_qty: 1, avg_price: 1,
		avg_disc: 1, count_order: 1
		}
		},
		{$sort: { l_returnflag: 1, l_linestatus: 1 }}
		])

Q4	<initial step in code to create the collection
	normalized_q4_new_joined_orders is ommited>
	
	db.normalized_q4_new_joined_orders.aggregate([
	{
	$project:{
	ORDERDATE:1,ORDERPRIORITY:1,
	eq:{
	$cond:[
	{$lt:["$lineitems.COMMITDATE",
	"$lineitems.RECEIPTDATE"]}
	,0
	,1
	]
	}
	}
	},
	{$match:{eq:{$eq:1}}},
	{
	$group:{
	_id:{ORDERPRIORITY:"$ORDERPRIORITY"},
	order_count:{$sum:1}
	}
	},
	{$sort:{ORDERPRIORITY:1}}
	])
		db.orders.aggregate([
		{
		$match: {
		ORDERDATE: {
		$gte: ISODate("[DATE]"),
		$lt: new Date(ISODate("[DATE]").getTime() +
		(3 * 30 * 24 * 60 * 60 * 1000))
		}
		}
		},
		{
		$lookup: {
		from: "lineitem",
		localField: "ORDERKEY",
		foreignField: "ORDERKEY",
		as: "lineitems"
		}
		},
		{
		$match: {
		$expr: {
		$lt: ["$lineitems.COMMITDATE", "$lineitems.RECEIPTDATE"]
		}
		}
		},
		{$group: {_id: "$ORDERPRIORITY", order_count: { $sum: 1 }}},
		{$sort: { _id: 1 }},
		{
		$project: {_id: 0, o_orderpriority: "$_id", order_count: 1}
		}
		])

Table 6: Translation of TPC-H SQL queries to MQL in the baseline and by ChatGPT 3.5

```
{
  $match: {
    lineitems: {
      $elemMatch: {
        COMMITDATE: { $lt: "$lineitems.RECEIPTDATE" }
      }
    }
  }
}
```

The iteration was to follow up with another prompt, which simply stated that "The match after lookup is not working". This resulted in the specification of the `$match` operation being corrected to:

```
{
  $match: {
    $expr: {
      $lt: ["$lineitems.COMMITDATE", "$lineitems.RECEIPTDATE"]
    }
  }
}
```

There were a couple of other issues: 1) the collection name was incorrect (`orders` instead of `order`), resulting from the table name in the SQL query; and 2) it is necessary to change the date filters in the first `$match`. With these simple changes, the final query was correct. The query implementation generated by ChatGPT used a significantly different strategy from the baseline, precluding the initial step of creating an intermediate collection by joining the `lineitem` and `order` collections in code, and only then calculating the final result over this collection. The baseline strategy constructs the intermediate collection by querying each document separately. The version generated by ChatGPT yields an implementation with better performance, again, avoiding the creation of an intermediate collection.

As with the TPC-C based tests (Section 4.2.1), the results were encouraging, given that even when the translated query was not immediately correct, very few iterations with the LLMs resulted in a correct query.

4.2.3 Cypher to SQL

With results from the translation from SQL to MQL being positive, let's now move on to a different query language. Cypher, which is Neo4j's query language, is suitable for graph related queries. Depending

on the queries, the same behavior of a Cypher query might not be simple to accomplish in SQL, which provides a great test for LLMs.

No.	SQL

1	`MATCH (p:Person)<-[:EMPLOYEE]-(d:Department)`
	`WHERE d.name = "IT Department"`
	`RETURN p.name`

2	`MATCH (c:Customer {companyName:'Drachenblut Delikatessen'})`
	`OPTIONAL MATCH (p:Product)<-[o:ORDERS]-(:Order)<-[:PURCHASED]-(c)`
	`RETURN`
	`  p.productName,`
	`  toInteger(sum(o.unitPrice * o.quantity)) AS volume`
	`ORDER BY volume DESC`

3	`MATCH (:Order)<-[:SOLD]-(e:Employee)`
	`WITH e, count(*) as cnt`
	`ORDER BY cnt DESC LIMIT 10`
	`RETURN e.employeeID, e.firstName, e.lastName, cnt`

4	`MATCH (u:Customer {customer_id:'customer-one'})-[:BOUGHT]`
	`->(p:Product)<- [:BOUGHT]-(peer:Customer)-[:BOUGHT]`
	`->(reco:Product)`
	`WHERE not (u)-[:BOUGHT]->(reco)`
	`RETURN reco as Recommendation, count(*) as Frequency`
	`ORDER BY Frequency DESC LIMIT 5`

5	`MATCH (p:Product)-[:PART_OF]->(l:Category)-[:PARENT*0..]`
	`-(:Category {name:'Dairy Products'})`
	`RETURN p.name`

Table 7: Cypher queries

After some iterations, we found that prompts of the form

```
Only output code. Convert the following Cypher query to a SQL query:
<Cypher query>
```

with <Cypher query> replaced with the target query, provided the most similar results to the ground truth.

Starting with query 1, the first row in Table 7 and Table 8, there was the need for a reprompt after the initial interaction with the LLM. The first output was:

```
SELECT p.name
FROM Person p
```

No.	SQL Baseline	SQL by ChatGPT

1	SELECT name FROM Person
	LEFT JOIN Person_Department
	  ON Person.Id = Person_Department.PersonId
	LEFT JOIN Department
	  ON Department.Id = Person_Department.DepartmentId
	WHERE Department.name = "IT Department"
		SELECT p.name FROM Person p
		JOIN Person_Department pd
		  ON p.id = pd.person_id
		JOIN Department d
		  ON pd.department_id = d.id
		WHERE d.name = 'IT Department'

2	SELECT p.ProductName, sum(od.UnitPrice * od.Quantity) AS Volume
	FROM customers AS c
	LEFT OUTER JOIN orders AS o ON (c.CustomerID = o.CustomerID)
	LEFT OUTER JOIN order_details AS od ON (o.OrderID = od.OrderID)
	LEFT OUTER JOIN products AS p ON (od.ProductID = p.ProductID)
	WHERE c.CompanyName = 'Drachenblut Delikatessen'
	GROUP BY p.ProductName
	ORDER BY Volume DESC
		SELECT p.productName,
		  CAST(SUM(o.unitPrice * o.quantity) AS INTEGER)
		  AS volume
		FROM Customer c
		LEFT JOIN Purchase pur ON pur.customerId = c.customerId
		LEFT JOIN Orders o ON o.orderId = pur.orderId
		LEFT JOIN Product p ON p.productId = o.productId
		WHERE c.companyName = 'Drachenblut Delikatessen'
		GROUP BY p.productName
		ORDER BY volume DESC

3	SELECT e.EmployeeID, e.FirstName, e.LastName, COUNT(*) AS Count FROM Employee AS e JOIN Orders AS o ON (o.EmployeeID = e.EmployeeID) GROUP BY e.EmployeeID, e.FirstName, e.LastName ORDER BY Count DESC LIMIT 10	SELECT e.employeeID, e.firstName, e.lastName, COUNT(*) AS cnt FROM Employee e JOIN Order o ON o.employeeID = e.employeeID GROUP BY e.employeeID, e.firstName, e.lastName ORDER BY cnt DESC LIMIT 10

4	SELECT product.product_name as Recommendation, count(1) as Frequency
	FROM product,
	  customer_product_mapping,
	  (SELECT cpm3.product_id, cpm3.customer_id
	  FROM Customer_product_mapping cpm,
	  Customer_product_mapping cpm2,
	  Customer_product_mapping cpm3
	  WHERE cpm.customer_id = 'customer-'one
	  AND cpm.product_id = cpm2.product_id
	  AND cpm2.customer_id != 'customer-'one
	  AND cpm3.customer_id = cpm2.customer_id
	  AND cpm3.product_id not in
	  (SELECT DISTINCT product_id
	  FROM Customer_product_mapping cpm
	  WHERE cpm.customer_id = 'customer-'one)
	  ) recommended_products
	WHERE customer_product_mapping.product_id
	  = product.product_id
	  AND customer_product_mapping.product_id
	  IN recommended_products.product_id
	  AND customer_product_mapping.customer_id
	  = recommended_products.customer_id
	GROUP BY product.product_name
	ORDER BY Frequency desc
		SELECT reco.product_id AS Recommendation, COUNT(*) AS Frequency
		FROM Customer u
		JOIN Bought b1 ON u.customer_id = b1.customer_id
		JOIN Product p ON b1.product_id = p.product_id
		JOIN Bought b2 ON p.product_id = b2.product_id
		JOIN Customer peer ON b2.customer_id = peer.customer_id
		JOIN Bought b3 ON peer.customer_id = b3.customer_id
		JOIN Product reco ON b3.product_id = reco.product_id
		WHERE u.customer_id = 'customer-one'
		  AND NOT EXISTS (
		  SELECT 1
		  FROM Bought b4
		  WHERE b4.customer_id = u.customer_id
		  AND b4.product_id = reco.product_id
		  )
		GROUP BY reco.product_id
		ORDER BY Frequency DESC
		LIMIT 5

5	SELECT p.ProductName
	FROM Product AS p
	JOIN ProductCategory pc
	  ON (p.CategoryID = pc.CategoryID
	  AND pc.CategoryName = "Dairy Products")
	JOIN ProductCategory pc1
	  ON (p.CategoryID = pc1.CategoryID)
	JOIN ProductCategory pc2
	  ON (pc2.ParentID = pc2.CategoryID
	  AND pc2.CategoryName = "Dairy Products")
	JOIN ProductCategory pc3
	  ON (p.CategoryID = pc3.CategoryID)
	JOIN ProductCategory pc4
	  ON (pc3.ParentID = pc4.CategoryID)
	JOIN ProductCategory pc5
	  ON (pc4.ParentID = pc5.CategoryID
	  AND pc5.CategoryName = "Dairy Products")
		WITH RECURSIVE CategoryHierarchy AS (
		  SELECT c1.id, c1.name, c1.parent_id
		  FROM Category c1
		  WHERE c1.name = 'Dairy Products'
		  UNION ALL
		  SELECT c2.id, c2.name, c2.parent_id
		  FROM Category c2
		  JOIN CategoryHierarchy ch ON c2.parent_id = ch.id
		)
		SELECT p.name
		FROM Product p
		JOIN CategoryHierarchy ch ON p.category_id = ch.id

Table 8: Cypher queries translation to SQL by baseline and by ChatGPT

```
JOIN Department d ON p.department_id = d.id
WHERE d.name = 'IT Department'
```

which does not take into account that each Department can have more than one Person. So, after re-prompting with:

```
Each Department can have more than one Person
```

the final result was achieved. We can right away see how similar they are. Apart from the column names, for example `PersonId` vs `person_id`, that ChatGPT has no way of knowing, the general structure of the query is correct. Also, the `JOIN` used is not the same, but this could be easily modified if needed.

Moving on to query 2, the translation ended up being very similar to the baseline. In fact, it can be said that it is closer to the Cypher query, as the SQL table names more closely match the Cypher labels. Either way, the behavior is functionally equivalent.

Regarding query 3, apart from slight naming differences, the queries are virtually identical, so there is no need to analyze in more detail.

For 4, there are major differences between the baseline and the translated query. But that does not mean it is wrong, as the translated query proved to be correct when tested and is arguably more concise. Also, the baseline missed the limit of five products imposed by the Cypher counterpart.

When it comes to query 5, the baseline only supports 3 levels of parent relationships for the category, instead of an arbitrary number as in Cypher. In the case of the query generated by ChatGPT, it correctly supports arbitrary category hierarchies by using recursive CTEs (Common table Expressions). In spite of this, it was necessary to reprompt, as the first version of the query was:

```
SELECT p.name
FROM Product p
JOIN Category l ON p.category_id = l.id
JOIN Category parent ON l.lft BETWEEN parent.lft AND parent.rgt
WHERE parent.name = 'Dairy Products'
```

which only supports one level of relationship. But when reprompted with:

```
Category hierarchies can be arbitrary
```

it generated the final correct version. This example clearly demonstrates that LLMs can aid in query translation even for esoteric queries.

In line with the other tests, we got very good results with regards to query translation using LLMs.

4.3 Implementation

The overall implementation follows the design presented in the proposal in Section 4.1, with the components developed in Java.

To ensure Byzantine fault tolerance, we chose BFT-SMaRt [29], a state machine replication library. It abstracts the mechanism of State Machine Replication, by employing a modular design approach that allows for easy customization and extension of the library's components, while achieving competitive performance compared to other BFT libraries. Also, by being implemented in Java, it aligns with our choice of language.

The architecture of BFT-SMaRt consists of several key components. At its core is a Byzantine fault-tolerant total order multicast protocol, which ensures that all non-faulty replicas deliver the same sequence of messages, maintaining consistency across the system. The library also implements an efficient state transfer protocol that allows new or recovering replicas to obtain the current system state from other replicas. Furthermore, BFT-SMaRt supports dynamic reconfiguration, enabling the addition or removal of replicas without interrupting the system's operation.

Moving to our system, its core is structured around a few key classes that manage communication between the client and the server, as well as between the system and the underlying databases. These classes are designed to provide as much flexibility as possible.

4.3.1 Request and Response Classes

HRRequest encapsulates the data necessary for a query to be executed. Each request consists of the query itself, which implements `QueryInterface`, and the query parameters. The parameters are of type `Map<String, Object>`, which allows for great flexibility. Multiple `HRRequest` objects are then bundled into a `HRRequests` object before being sent to the server, which allows for batching.

On the other end, **HRResponse** represents the result of a query execution. It is structured as a `List` of `TreeMap<String, Object>` entries, allowing for a flexible structure, which should accommodate the majority of data models. This structure can be thought of as a sequence of key-value mappings. The use of `TreeMap` ensures that the order of the key-value entries is maintained, which is a requirement of BFT-SMaRt's state transfer protocol. Like the requests, multiple `HRResponse` objects are bundled into `HRResponses` before being returned to the client, each corresponding to a request.

4.3.2 Query and Driver Interfaces

The support for arbitrary database models is achieved through the use of interfaces that define the required behaviors for queries and database drivers.

QueryInterface defines two methods that all queries must implement. `getQuery` returns the query string to be executed given a specific driver. `getParameters` returns a `List` with the names of the parameters required for the query, aiming at preventing the execution of queries with missing parameters. This interface ensures that all queries are handled in a uniform way, regardless of the underlying database.

Similarly, **DriverInterface** defines the interaction with a database. This interface includes a single method `executeRequest`, which accepts an `HRRequest` and returns an `HRResponse`. There are currently four implementations of the `DriverInterface`, supporting the databases MongoDB [43], MySQL [88], Neo4j [46] and PostgreSQL [89].

4.3.3 Driver Implementations

Each database driver implements the `DriverInterface`, with the specific logic varying depending on the database being used.

The drivers for `MySQL` and `PostgreSQL` are very similar and share most of the implementation. The main difference is the underlying JDBC driver used: the **MySQLDriver** uses `com.mysql.cj.jdbc.Driver` and the **PostgreSQLDriver** uses `org.postgresql.Driver`. In spite of not being necessary for our tests, the implementations were kept separated to allow for different implementations of queries, given that there are differences between their dialects of SQL. In order to execute a query, it starts by getting the query string for the corresponding database and then using an helper method `getQueryAndOrderedParams` to extract the query with placeholders and the `List` of the parameters as they appear in the query. After that, it uses a prepared statement from the underlying database driver and adds the parameters to it. Finally, it executes the prepared statement and evaluates whether it was a select query or not. If it was, a result is constructed in the `List<TreeMap<String, Object>>` format and returned as a response. Otherwise, the update count is returned in a structure compatible with the above format.

When it comes to the **MongoDBDriver**, it starts by getting the corresponding query string and then calling the same helper function `getQueryAndOrderedParams`. Where it differs is that the parameters are replaced with the actual value. Particularly, in the case of Timestamp's, they are converted to the string `new ISODate('<timestamp>')`. The list of supported queries can be found in the MongoDB documentation [90]. Also, for updates that can apply to multiple documents, there is the need to add the

option `multi: true`. The query is then executed and, if the returned document contains a cursor, the values are extracted and new queries are made to extract more values until done, always converting them to the format previously discussed. Otherwise, it behaves as the previous drivers.

In the case of the **Neo4jDriver**, the driver uses the query string and the parameters directly. After executing, it returns a list of records that are then converted into the discussed format. Like the other drivers, the Neo4j driver handles select and non-select queries differently.

4.3.4 Server Class

The server-side component, **HRServer**, extends the `DefaultRecoverable` class from BFT-SMaRt, which keeps the state in memory, but can optionally persist it to disk. This class manages the state of the server and processes client requests in batches of `HRRequests`. `HRServer` is configured using a unique identifier and the connection information for the appropriate database. The system supports fault tolerance by running multiple replicas, and the BFT-SMaRt protocol handles consensus and state synchronization across these replicas. Additionally, `HRServer` can retrieve and install snapshots, using the database drivers to extract or restore data during state transfer operations.

4.3.5 Client Class

On the client-side, the **HRClient** class incorporates a `ServiceProxy` object from BFT-SMaRt, which provides an API for interacting with the servers. The `HRClient` also handles Byzantine fault detection by comparing responses from the various replicas. If an inconsistent response is detected, the `HRClient` sends a shutdown request to the cluster, triggering the restart of the faulty server, which triggers the state recovery mechanism via snapshot transfer from the correct replicas. This ensures that even if one replica fails, the system can maintain its consistency and correct behavior.

4.3.6 Query Translation

The queries executed by the system were initially written in SQL and then automatically converted to the corresponding dialects for MongoDB and Neo4j. This conversion process was aided by Codeium [91], an autocompletion tool that can be integrated with the code editor. In some cases, ChatGPT was used to refine the queries, addressing issues that arose during testing. In rare cases, manual intervention was required to check the official documentation and correct specific query errors.

4.4 Takeaways

The research and implementation described in this chapter yield several important contributions:

1. **Proposed Architecture:** The proposed architecture allows for the use of diverse database systems within a single replicated environment. This heterogeneity enhances fault tolerance by mitigating the risk of common vulnerabilities across replicas and leverages a broad spectrum of existing databases across several different models. We also avoid the need for a middleware layer, which reduces the surface for errors.

2. **Viability of LLMs for Query Translation:** We showed with practical examples that LLMs are suitable for query translation as needed by our architecture. Our solution can also take advantage of progresses made in LLMs, potentially unlocking live translations in the future. In spite of all these tests, one of the main open research issues is to experiment with query translation on a larger scale, by using a wider variety of queries and target database systems. In fact, such experiments are needed to overcome a threat to the validity of the experiments presented that stems from using standard benchmarks, for which queries exist online, and thus certainly in the training set for the LLMs that were used.

3. **Implementation:** An implementation of the proposed architecture was presented, paving the way for future contributions in the area of heterogeneous replication that could take it as a starting point. It achieves a level of support for different database models that can't be found in previous work.

Chapter 5

Evaluation

This chapter presents a comprehensive evaluation of our system, consisting of two parts: functional testing to verify correctness and system behavior, and performance benchmarking to assess the system's efficiency and scalability under various workloads.

Our testing methodology progresses from basic database operations to industry-standard benchmarks, including the implementation of TPC-C and YCSB workloads. These tests not only validate the system's functionality but can also help find issues with the initial versions of the implementation and surface important insights about the challenges and solutions in heterogeneous database replication.

The performance evaluation examines our system under different configurations and workloads, comparing it against traditional database deployments. This analysis provides valuable insights into the performance trade-offs inherent in our approach, particularly regarding the balance between enhanced fault tolerance and system throughput.

Through these evaluations, we aim to demonstrate both the viability of our heterogeneous replication approach and its practical implications for real-world deployments.

5.1 Testing

Following the development of our system, we conducted a series of comprehensive tests to validate our design decisions and highlight areas for potential future improvements.

5.1.1 Basic CRUD Operations Test

Our initial test focused on simple Create, Read, Update, and Delete (CRUD) operations. We utilized a structure containing a single integer field named `id` for this purpose. The test procedure involved a sequence of operations: deleting all existing rows, inserting a single row, selecting to confirm insertion, updating the value, selecting again to verify the update, and finally deleting the row.

For instance, the insert operation was implemented as follows:

- MySQL/PostgreSQL:

```
INSERT INTO test (id) VALUES ($id)
```

- Neo4j:

```
CREATE (n:test {id: $id}) RETURN COUNT(n) AS count
```

- MongoDB:

```
{insert: 'test', documents: [{id: $id}]}
```

The select operation was implemented as follows:

- MySQL/PostgreSQL:

```
SELECT id FROM test ORDER BY id ASC
```

- Neo4j:

```
MATCH (n:test) RETURN n.id AS id ORDER BY n.id ASC
```

- MongoDB:

```
{find: 'test', projection: {id: 1}, sort: {id: 1}}
```

The update operation was implemented as follows:

- MySQL/PostgreSQL:

```
UPDATE test SET id = $setId WHERE id = $whereId
```

- Neo4j:

```
MATCH (n:test)
WHERE n.id = $whereId SET n.id = $setId
RETURN COUNT(n) AS count
```

- MongoDB:

```
{update: 'test',
 updates: [{q: {id: $whereId}, u: {$set: {id: $setId}}}]}
```

The delete operation was implemented as follows:

- MySQL/PostgreSQL:

```
DELETE FROM test
```

- Neo4j:

```
MATCH (n:test) DETACH DELETE n RETURN COUNT(n) AS count
```

- MongoDB:

```
{delete: 'test', deletes: [{q: {}, limit: 0}]}
```

In these queries, that are two important details to consider: for Read queries, the implementation uses a sorting clause; for Create, Update and Delete queries, the implementation returns the number of affected rows (in some of the databases this is implicit).

Regarding the automated translations that generated these queries, they required minimal intervention, with the translation being lead by LLMs. These first tests surfaced some bugs that ended up being easy to fix.

5.1.2 Incremental Update Test

The second phase of testing focused on evaluating the system's correctness with incremental updates on top of the same table as the previous test. This test began with an optional deletion of all existing rows, followed by the insertion of two initial values: 1 and 10. Subsequently, we performed 1,000 incremental updates on these values, with a select operation after each increment to verify the changes.

The increment operation was implemented as follows:

- MySQL/PostgreSQL:

```
UPDATE test SET id = id + 1
```

- Neo4j:

```
MATCH (n:test)
SET n.id = n.id + 1
RETURN COUNT(n) AS count
```

- MongoDB:

```
{update: 'test',
 updates: [{q: {}, u: {$inc: {id: 1}},
 multi: true}]}
```

The remaining queries (select, insert and delete) were exactly the same as in the previous test. As in the previous tests, the translations required minimal intervention.

5.1.3 TPC-C Benchmark Implementation

To provide a standardized measure of our system's correctness, we implemented the TPC-C benchmark [92] from BenchBase [93]. This implementation required some adaptations from the base implementation to ensure compatibility with our system architecture.

The modifications included:

1. Adapting the calls to the database to instead use HRClient.
2. Reducing the batch size to 64, as otherwise the limit of the system was being reached.
3. Altering the MySQL DDL by changing timestamp columns from `timestamp` to `timestamp(3)` and `float` to `double` for improved compatibility with the other databases. Regarding data compatibility, there is always the option to serialize incompatible types to string or bytes, as they are supported by most databases. Then, in the application, they would be converted back to the intended type. In our case, we had no need to do this.
4. Modifying the PostgreSQL `ol_quantity` column from `decimal(6,2)` to `int` to maintain compatibility with the other databases.

Rather than developing a full JDBC driver, which would have been a significant undertaking, we opted to send requests using the previously described HRClient. This decision allowed us to simplify the integration while still leveraging the comprehensive nature of the TPC-C benchmark. But keep in mind that there is nothing preventing the implementation of said driver.

Most queries were simple, very close to the previous tests, so we will limit the queries presented to a couple that deserve special attention.

First, is the query to calculate the total value of an order:

- MySQL/PostgreSQL:

```
SELECT SUM(ol_amount) AS ol_total
FROM order_line
WHERE ol_o_id = $olOId
  AND ol_d_id = $olDId
  AND ol_w_id = $olWId
```

- Neo4j:

```
MATCH (n:order_line)
WHERE n.ol_o_id = $olOId
  AND n.ol_d_id = $olDId
  AND n.ol_w_id = $olWId
RETURN sum(n.ol_amount) AS ol_total
```

- MongoDB:

```
{aggregate: 'order_line',
 pipeline: [
   {$match: {
      ol_o_id: $olOId,
      ol_d_id: $olDId,
      ol_w_id: $olWId
   }},
   {$group: {
      _id: null,
      ol_total: {$sum: '$ol_amount'}
   }}
 ],
 cursor: {}}
```

Then, a query that determines the number of recently sold items that have a stock level below a specified threshold.

- MySQL/PostgreSQL:

```
SELECT COUNT(DISTINCT(s_i_id)) AS stock_count
FROM order_line, stock
WHERE ol_w_id = $olWId
  AND ol_d_id = $olDId
  AND ol_o_id < $olOIdLess
  AND ol_o_id >= $olOIdGreaterEqual
  AND s_w_id = $sWId
  AND s_i_id = ol_i_id
  AND s_quantity < $sQuantity
```

- Neo4j:

```
MATCH (ol:order_line), (s:stock)
WHERE ol.ol_w_id = $olWId
  AND ol.ol_d_id = $olDId
  AND ol.ol_o_id < $olOIdLess
  AND ol.ol_o_id >= $olOIdGreaterEqual
  AND s.s_w_id = $sWId
  AND s.s_i_id = ol.ol_i_id
  AND s.s_quantity < $sQuantity
RETURN COUNT(DISTINCT(s.s_i_id)) AS stock_count
```

- MongoDB:

```
{aggregate: 'order_line',
 pipeline: [
   {$match: {
      ol_w_id: $olWId,
      ol_d_id: $olDId,
      ol_o_id: {$lt: $olOIdLess, $gte: $olOIdGreaterEqual}
   }},
   {$lookup: {
      from: 'stock',
      localField: 'ol_i_id',
```

```
            foreignField: 's_i_id',
            as: 'stock'
        }},
        {$unwind: '$stock'},
        {$facet: {matched: [{
            $match: {
              'stock.s_w_id': $sWId,
              'stock.s_quantity': { $lt: $sQuantity }
            }
        }]}},
        {$project: {stock_count: {$size: '$matched'}}}
    ],
    cursor: {}}
```

Contrasting to the previous examples, these queries proved harder to be translated correctly. In the case of Neo4j, given the similarity in syntax to SQL, there were no issues and the right query was generated. When it came to MongoDB, it required going to ChatGPT and consulting the documentation. In spite of this, the use of Codeium and ChatGPT allowed for a faster iteration on the query, requiring minimal consultation of the documentation, which would otherwise take much longer without these tools. After each query generation, the system was tested to validate if the query was returning the correct results when compared to the other databases.

5.1.4 YCSB Benchmark Implementation

We also implemented the YCSB benchmark [94] from BenchBase [93]. In contrast to TPC-C, no major change was required. We also opted to not implement a full JDBC driver.

This implementation actually revealed a problem in the way we were handling UTF-8 strings in MongoDB. The fix was to convert their characters to the corresponding code points before sending them to the database.

As in TPC-C, most queries are simple, but the query to select the maximum key from `usertable` stands out:

- MySQL/PostgreSQL:

```
SELECT MAX(ycsb_key) AS max_ycsb_key
FROM usertable
```

- Neo4j:

```
MATCH (n:usertable)
RETURN MAX(n.ycsb_key) AS max_ycsb_key
```

- MongoDB:

```
{aggregate: 'usertable',
 pipeline: [{
   $group: {
     _id: null,
     max_ycsb_key: {$max: '$ycsb_key'}
   }
 }],
 cursor: {}}
```

During the translation of queries for this benchmark, most translations were immediate. The only case where this didn't happen was in the query above, but the correct version of the query was quickly developed after consulting with ChatGPT.

5.1.5 Test Outcomes

The execution of these tests proved invaluable not only for evaluating the system behavior but also for identifying and rectifying several bugs. These issues primarily related to discrepancies in query behavior across different database systems, validating our initial assumption that such adaptations would be necessary given the heterogeneous nature of our architecture.

The process of resolving these bugs has contributed significantly to the robustness and reliability of our system. It has also provided valuable insights into the details related to the integration of multiple database technologies within a single system.

5.2 Benchmarking

To evaluate the performance of our system, we conducted a series of benchmarks based on the adapted TPC-C and YCSB workloads described in Section 5.1. The benchmarks were performed with default settings, which means a single warehouse for TPC-C and 1000 records for YCSB. Additionally, they were run with 1 and 10 terminals to test the scalability in the presence of concurrent clients.

5.2.1 Benchmark Configurations

We tested five different database configurations:

A. **Single PostgreSQL:** A single PostgreSQL 16.4 database running with default settings.

B. **PostgreSQL Replication:** Two PostgreSQL 16.4 databases running streaming synchronous replication with default settings, where one is the primary and the other the standby.

C. **Our System - Consensus Only:** Our system running in a configuration of four BFT instances without an underlying database, aiming at testing the overhead of the consensus protocol.

D. **Our System - Homogeneous:** Our system running in a homogeneous replication configuration of four BFT instances, with each of them managing a PostgreSQL 16.4 database, aiming at demonstrating that our system supports the same configurations as previous homogeneous solutions.

E. **Our System - Heterogeneous:** Our system running in a heterogeneous replication configuration of four BFT instances, with each of them managing a PostgreSQL 16.4, MySQL 8.4.2, Neo4j 4.4.37, or MongoDB 7.0.13 database.

Regarding the performance results of configurations D and E, any difference between them can solely be attributed to the speed of the underlying databases, as they are otherwise identical.

5.2.2 Hardware Configurations

The benchmarks were run on two different machine configurations:

1. **Single-host Deployment:** One C3-32 server from OVHcloud [95] with the following specifications:

   16 vCores (2.3 GHz); 32 GB RAM; 400 GB NVMe; 2000 Mbit/s; Ubuntu 24.04; located in Gravelines, France (GRA11)

2. **Multi-host Deployment:** Five `AX41-NVME` dedicated servers from Hetzner [96] with the following specifications:

   6 cores (3.6GHz, AMD Ryzen™ 5 3600); 64GB RAM; 2 x 512 GB NVMe SSD (software-RAID 1); 1 GBit/s, 0.5 ms latency between servers; Ubuntu 24.04; located in Helsinki, Finland (HEL1)

In configuration 1, the BFT instances, clients and databases were run in a single machine. It allowed us to test without the effects of network latency but had the drawback of resulting in contention for machine resources, which skewed some of the results for this configuration. That is because configurations A and B have proportionally more resources. To be more specific, the database for configuration A only needs to share the resources with the benchmark clients, while in configuration D and E, the resources are shared between the four databases, four BFT instances and the benchmark clients. That said, the results will be presented but not discussed, as they are not representative of a real world fault tolerant deployment, where each instance will run on its own server.

In configuration 2, each of the BFT instances and the clients were run in a different server. In contrast to the previous configuration, it allowed to add a small quantity of latency and discard any possible contention. This is more in line with a real world deployment, as it keeps everything separated and improves fault tolerance.

5.2.3 Results

The results below are presented in requests per second of throughput, as reported by the BenchBase tool. Each test was run 5 times and any outlier was discarded before averaging the results.

TPC-C

Configuration	1 Terminal	10 Terminals
A (Single PostgreSQL)	443	1304
B (PostgreSQL Replication)	328	1115
C (Our System - Consensus Only)	55	379
D (Our System - Homogeneous)	30	105
E (Our System - Heterogeneous)	14	24

Table 9: TPC-C throughput for the Single-host Deployment (in req/s)

Configuration	1 Terminal	10 Terminals
A (Single PostgreSQL)	117	519
B (PostgreSQL Replication)	103	419
C (Our System - Consensus Only)	33	253
D (Our System - Homogeneous)	20	51
E (Our System - Heterogeneous)	11	17

Table 10: TPC-C throughput for the Multi-host Deployment (in req/s)

For TPC-C, Table 9 presents the throughput results (in requests per second) for the Single-host Deployment configuration and Table 10 for the Multi-host Deployment configuration.

Regarding the Multi-host Deployment results, from configuration A to B there was a decrease in performance, which is expected, given that synchronous replication requires the standby replica to apply the changes before a transaction is committed. We will compare our system to B as it is more fault tolerant than A, where no replication is used.

Now, when testing the overhead of consensus in configuration C in relation to configuration B, we get a 67.96% and 39.62% decrease in performance, which is even greater in configurations D and E. An important thing to mention is that the execution of queries has an impact on scalability, as shown by the scalability of configuration C from 1 to 10 terminals when compared to D and E.

Depending on the use case, this performance overhead might be acceptable, because it ends up being a trade-off between performance and fault tolerance. That is, configuration A is a single point of failure and B can't handle the failures related to vulnerabilities we previously discussed.

An important note is that there is a backlog of queries after the benchmark runs finish. BFT-SMaRt, for the failure mode being considered where 1 faulty replica is tolerated, requires 2 agreeing responses to reply to the client. That is because after two responses agree with each other, it means the execution order has been determined and one of the replicas can be assumed to be healthy. On the load phase of the benchmark, PostgreSQL and MongoDB lead the queries execution, showing they provide the best insert performance. On the execute phase, PostgreSQL and MySQL lead the queries execution, which makes sense as the TPC-C is an OLTP benchmark, which is tailored for relational databases.

Configuration	1 Terminal	10 Terminals
A (Single PostgreSQL)	4607	28444
B (PostgreSQL Replication)	3881	26713
C (Our System - Consensus Only)	950	5789
D (Our System - Homogeneous)	229	1542
E (Our System - Heterogeneous)	218	1044

Table 11: YCSB throughput for the Single-host Deployment (in req/s)

Configuration	1 Terminal	10 Terminals
A (Single PostgreSQL)	847	6830
B (PostgreSQL Replication)	743	6470
C (Our System - Consensus Only)	570	3936
D (Our System - Homogeneous)	129	611
E (Our System - Heterogeneous)	112	411

Table 12: YCSB throughput for the Multi-host Deployment (in req/s)

YCSB

For YCSB, Table 11 presents the throughput results (in requests per second) for the Single-host Deployment configuration and Table 12 for the Multi-host Deployment configuration.

When it comes to Multi-host Deployment results, this benchmark presents a notably different performance profile compared to TPC-C. From configuration B to C there is a performance decrease of 23.28% for 1 terminal and 39.17% for 10 terminals, which, when compared to TPC-C, reveals a smaller impact in the face of low concurrency and the same impact when it increases. As in the case of TPC-C, our solution does not scale as well when executing queries, given the serial execution required for correctness.

As shown, the performance characteristics of the system depend greatly on the workload, which is a consideration to be taken into account when deciding if it is a viable solution.

Performance Trade-off

While our system shows lower performance compared to traditional deployments, these results must be contextualized within the broader system objectives. The performance decrease in Configurations D and

E represents the cost of achieving higher data safety and fault tolerance. This trade-off may be acceptable for applications prioritizing these characteristics over performance, which is the case for mission critical workloads.

The benchmark results also highlight areas for potential optimization, particularly in reducing the base system overhead demonstrated in Configuration C and in the concurrent execution of queries. Future work could focus on minimizing these overheads while maintaining the system's architectural advantages.

5.3 Takeaways

The evaluation of our heterogeneous database replication system revealed several important takeaways:

1. **Testing:** By testing our implementation, it was shown that our solution is viable for existing workloads such as TPC-C and YCSB. It was also an import step in the development of the system, as it uncovered multiple bugs that would otherwise go unnoticed.

2. **Benchmarking:** By benchmarking our solution, we revealed there is a performance impact from the utilization of our solution. Despite that, based on the use cases being discussed, this trade-off might be worth considering, as our solution provides greater fault tolerance and lower probability of data mishandling due to vulnerabilities.

This suggests that while our approach introduces new capabilities in terms of database diversity and fault tolerance, there is a need for further research and optimization to make it more competitive in performance-critical applications. The system as it stands may be most suitable for use cases where the benefits of strong fault tolerance outweigh the need for maximum throughput.

Furthermore, the proposed architecture and its implementation show it is possible to have a heterogeneous system that tolerates Byzantine faults, while integrating databases with different models, which in our tests contemplated relational, graph and document databases.

Chapter 6

Conclusions and Future Work

This dissertation proposed a novel heterogeneous database replication architecture that is viable and presents meaningful advantages, particularly in fault tolerance and system resilience. To support it, we presented an empirical study that revealed the threat of software vulnerabilities and the insufficiency of homogeneous database replication, showing the potential of heterogeneous replication for dependable database systems. The results showed that no shared vulnerabilities exist across different databases, supporting the feasibility of heterogeneous replication as a safeguard against security flaws. Additionally, the variety among open-source databases in terms of programming languages and data models further validates that modern database ecosystems can support such a replication model without compromising diversity.

By implementing LLMs for query translation, the system demonstrated promising results, proving that translation accuracy and efficiency can bridge the gap between different data models. This innovation eliminated the need for a separate, complex middleware layer, which typically introduces operational overhead and a higher likelihood of shared bugs across replicas. The viability of LLMs was demonstrated with query translations across SQL, MQL, and Cypher, with great results for a variety of cases.

The evaluation of our system covered both functional correctness and performance aspects, with results validating its ability to handle workloads with resilience and accuracy. Using standard benchmarks like TPC-C and YCSB further validated the system's robustness and practical application potential. Results of these benchmarks highlighted the trade-offs between enhanced fault tolerance and system throughput, a crucial consideration for adopting heterogeneous replication in real-world scenarios. While the system introduces performance overhead, it compensates by significantly enhancing reliability and operational continuity across multiple database models. This advantage is particularly relevant for critical applications where downtime or data loss are unacceptable.

Moreover, we also identify the following promising research directions:

Custom LLMs. It is interesting to use models that have been tailored to programming, and specifically those that have been already designed to manipulate database query languages. Moreover, it might be useful to customize models specifically for this task [97].

Testing. The proposed approach heavily relies on testing to validate that translated queries are, in fact, equivalent when applied to their corresponding databases. This is actually an advantage of our proposal, as testing is likely to be easier than the translation itself. Thus, it is interesting to take advantage of the n-version during testing, by comparing them, and to explore the possibility of automatically generating the tests, for instance, by using property-based testing [98, 99].

Query optimization. The proposed approach can leverage existing and future work on LLM-based query optimization for the target language in the query translation step. The result of the test with TPC-H's Q4 in Subsection 4.2.2 already serves as an example.

Online translation. The proposed approach is based on an offline step, during development, to obtain translations for application queries. It would however be interesting to research the possibility of moving this step online, so that ad-hoc queries can be used. However, this would raise the issue of model dependability and the availability of various LLMs [100]. Given enough redundancy, it might even be possible to cope with erroneous translations and recover, using reinforcement learning to improve the models themselves.

Direct access for read-only queries. It is possible to access the underlying databases directly for read-only queries. Depending on the data access pattern, it can be beneficial to use a database with a certain model.

Support for transactions. An interesting area of research is the support for transactions. In the current system, queries can only be executed individually, but it would be interesting to adapt the system to support database transactions, taking into account that each database behaves differently in this regard.

System optimization. If the system could be optimized to have a smaller impact on performance, it could unlock a broader set of applications, instead of only mission critical data, which is already a major potential application for our system.

Ease of use. Currently, the deployment of the system relies on manual intervention. An area of research would be on improving the deployment simplicity to help DBMS administrators take advantage of it. Also,

it would be an interesting undertaking to develop a JDBC driver that encapsulates the interaction with our system. It would simplify the integration into existing Java applications that are already using JDBC.

In conclusion, our research demonstrates the feasibility and benefits of heterogeneous database replication as a robust solution to modern database fault tolerance challenges. By introducing LLM-assisted query translation, we presented a more resilient and adaptable database architecture capable of supporting diverse data models.

Bibliography

[1] Database Definition. https://www.oracle.com/database/what-is-database/#database-defined, 2024.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[3] Peter Revesz. Introduction to databases. *Texts in Computer Science*, 2010.

[4] SQLite Stats, 2024. URL https://www.sqlite.org/mostdeployed.html.

[5] Jonas Wäfler and Poul E Heegaard. Interdependency modeling in smart grid and the influence of ict on dependability. In *Meeting of the European Network of Universities and Companies in Information and Communication Engineering*, pages 185–196. Springer, 2013.

[6] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 44(6):735–770, 2014.

[7] Caio Yuri da Silva Costa and Eduardo Adilio Pelinson Alchieri. Diversity on state machine replication. In *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, pages 429–436. IEEE, 2018.

[8] Rafael R Obelheiro, Alysson Neves Bessani, Lau Cheuk Lung, and Miguel Correia. How practical are intrusion-tolerant distributed systems? 2006.

[9] Ilir Gashi, Peter Popov, and Lorenzo Strigini. Fault tolerance via diversity for off-the-shelf products: A study with sql database servers. *IEEE Transactions on Dependable and Secure Computing*, 4(4): 280–294, 2007.

[10] Hiroyuki Yamada and Jun Nemoto. Scalar dl: Scalable and practical byzantine fault detection for transactional database systems. *Proceedings of the VLDB Endowment*, 15(7):1324–1336, 2022.

[11] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 739–752, 2008.

[12] Joao Rafael Henriques and José D'Abruzzo Pereira. Automated software vulnerability collection for a database with static information. 2023.

[13] CVEdetails.com, 2024. URL https://www.cvedetails.com.

[14] DB-Engines, 2024. URL https://db-engines.com.

[15] Jason L Wright, Jason W Larsen, and Miles McQueen. Estimating software vulnerabilities: A case study based on the misclassification of bugs in mysql server. In *2013 International Conference on Availability, Reliability and Security*, pages 72–81. IEEE, 2013.

[16] Nesara Dissanayake, Asangi Jayatilaka, Mansooreh Zahedi, and M Ali Babar. Software security patch management-a systematic literature review of challenges, approaches, tools and practices. *Information and Software Technology*, 144:106771, 2022.

[17] Nesara Dissanayake, Mansooreh Zahedi, Asangi Jayatilaka, and Muhammad Ali Babar. Why, how and where of delays in software security patch management: An empirical investigation in the healthcare sector. *Proceedings of the ACM on Human-computer Interaction*, 6(CSCW2):1–29, 2022.

[18] Oracle GoldenGate, 2024. URL https://www.oracle.com/integration/goldengate.

[19] Striim, 2024. URL https://www.striim.com.

[20] Continuent TungstenReplicator, 2024. URL https://www.continuent.com/products/tungsten-replicator.

[21] Jumpmind SymmetricDS, 2024. URL https://symmetricds.org.

[22] Tapdata, 2024. URL https://tapdata.io.

[23] Yongdong Wang and Jane Chiao. Data replication in a distributed heterogeneous database environment: An open system approach. In *Proceeding of 13th IEEE Annual International Phoenix Conference on Computers and Communications*, page 315. IEEE, 1994.

[24] Paulo Jorge Botelho Ferreira. Diverse intrusion-tolerant database replication. Master's thesis, Universidade de Lisboa (Portugal), 2012.

[25] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating Systems Principles*, pages 59–72, 2007.

[26] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1): 11–33, 2004.

[27] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.

[28] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[29] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[30] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[31] Xin Wang, Sisi Duan, James Clavin, and Haibin Zhang. Bft in blockchains: From protocols to use cases. *ACM Computing Surveys (CSUR)*, 54(10s):1–37, 2022.

[32] Gengrui Zhang, Fei Pan, Yunhao Mao, Sofia Tijanic, Michael Dang'Ana, Shashank Motepalli, Shiquan Zhang, and Hans-Arno Jacobsen. Reaching consensus in the byzantine empire: A comprehensive review of bft consensus algorithms. *ACM Computing Surveys*, 56(5):1–41, 2024.

[33] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, volume 1, pages 3–9, 1978.

[34] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. Efficient middleware for byzantine fault tolerant database replication. In *Proceedings of the sixth conference on Computer systems*, pages 107–122, 2011.

[35] Marcel Henrique dos Santos. Dependable data storage with state machine replication. Master's thesis, Universidade de Lisboa (Portugal), 2014.

[36] Ryan Marcus. Towards adaptive fault-tolerant sharded databases (extended abstracts). *Transactions*, 200:400, 2023.

[37] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 206–215. IEEE, 2000.

[38] Itamar Ankorion. Change data capture efficient etl for real-time bi. *Information Management*, 15 (1):36, 2005.

[39] PostgreSQL Foreign Data Wrappers. PostgreSQL foreign data wrappers. https://wiki.postgresql.org/wiki/Foreign_data_wrappers, 2024.

[40] Hugo Abreu, Luis Meruje Ferreira, Fábio Coelho, Ana Nunes Alonso, and José Pereira. Recovery in clouddbappliance's high-availability middleware. In *DATA*, pages 447–453, 2019.

[41] Fábio Coelho, Ana Alonso, Luis Ferreira, Jose Pereira, and Rui Oliveira. Loom: A closed-box disaggregated database system. In *Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing*, pages 30–39, 2023.

[42] Michael Stonebraker and Uğur Çetintemel. " one size fits all" an idea whose time has come and gone. In *Making databases work: the pragmatic wisdom of Michael Stonebraker*, pages 441–462. 2018.

[43] MongoDB. https://www.mongodb.com/, 2024.

[44] HBase. Apache hbase reference guide. https://hbase.apache.org/book.html, 2024.

[45] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review*, 44(2):35–40, 2010.

[46] Neo4j. https://neo4j.com/, 2024.

[47] Felix Kiehn, Mareike Schmidt, Daniel Glake, Fabian Panse, Wolfram Wingerath, Benjamin Wollmer, Martin Poppinga, and Norbert Ritter. Polyglot data management: state of the art & open challenges. *Proceedings of the VLDB Endowment*, 15(12):3750–3753, 2022.

[48] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The bigdawg polystore system. *ACM Sigmod Record*, 44(2):11–16, 2015.

[49] Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jiménez-Peris, Raquel Pau, and José Pereira. Cloudmdsql: querying heterogeneous cloud data stores with a common language. *Distributed and parallel databases*, 34:463–503, 2016.

[50] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*, 2023.

[51] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[52] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[53] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2. Minneapolis, Minnesota, 2019.

[54] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[55] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[56] Amadou Latyr Ngom and Tim Kraska. Mallet: Sql dialect translation with llm rule generation. In *Proceedings of the Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–5, 2024.

[57] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

[58] Zijin Hong, Zheng Yuan, Hao Chen, Qinggang Zhang, Feiran Huang, and Xiao Huang. Knowledge-to-sql: Enhancing sql generation with data expert llm. *arXiv preprint arXiv:2402.11517*, 2024.

[59] Zhenwen Li and Tao Xie. Using llm to select the right sql query from candidates. *arXiv preprint arXiv:2401.02115*, 2024.

[60] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. Evaluating the text-to-sql capabilities of large language models. *arXiv preprint arXiv:2204.00498*, 2022.

[61] Samuel Arcadinho, David Aparício, Hugo Veiga, and António Alegria. T5ql: Taming language models for sql generation. *arXiv preprint arXiv:2209.10254*, 2022.

[62] Richard Roberson, Gowtham Kaki, and Ashutosh Trivedi. Analyzing the effectiveness of large language models on text-to-sql synthesis. *arXiv preprint arXiv:2401.12379*, 2024.

[63] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*, 2023.

[64] Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. Enhancing text-to-sql capabilities of large language models: A study on prompt design strategies. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14935–14956, 2023.

[65] Ursin Brunner and Kurt Stockinger. Valuenet: A natural language-to-sql system that learns from database information. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2177–2182. IEEE, 2021.

[66] Qinggang Zhang, Junnan Dong, Hao Chen, Wentao Li, Feiran Huang, and Xiao Huang. Structure guided large language model for sql generation. *arXiv preprint arXiv:2402.13284*, 2024.

[67] Bin Zhang, Yuxiao Ye, Guoqing Du, Xiaoru Hu, Zhishuai Li, Sun Yang, Chi Harold Liu, Rui Zhao, Ziyue Li, and Hangyu Mao. Benchmarking the text-to-sql capability of large language models: A comprehensive evaluation. *arXiv preprint arXiv:2403.02951*, 2024.

[68] Federico Ranaldi, Elena Sofia Ruzzetti, Dario Onorati, Leonardo Ranaldi, Cristina Giannone, Andrea Favalli, Raniero Romagnoli, and Fabio Massimo Zanzotto. Investigating the impact of data contamination of large language models in text-to-sql translation. *arXiv preprint arXiv:2402.08100*, 2024.

[69] Fei Li and Hosagrahar V Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[70] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R Mittal, and Fatma Özcan. Athena: an ontology-driven system for natural language querying over relational data stores. *Proceedings of the VLDB Endowment*, 9(12):1209–1220, 2016.

[71] Vanessa Câmara, Rayol Mendonca-Neto, André Silva, and Luiz Cordovil. A large language model approach to sql-to-text generation. In *2024 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–4. IEEE, 2024.

[72] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. Natural language to sql: Where are we today? *Proceedings of the VLDB Endowment*, 13(10):1737–1750, 2020.

[73] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 2023.

[74] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

[75] Immanuel Trummer. Demonstrating gpt-db: Generating query-specific and customizable code for sql processing with gpt-4. *Proceedings of the VLDB Endowment*, 16(12):4098–4101, 2023.

[76] Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. Finsql: Model-agnostic llms-based text-to-sql framework for financial analysis. *arXiv preprint arXiv:2401.10506*, 2024.

[77] Ali Reza Ibrahimzada. Program decomposition and translation with static analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 453–455, 2024.

[78] Artur Tarassow. The potential of llms for coding with low-resource and domain-specific programming languages. *arXiv preprint arXiv:2307.13018*, 2023.

[79] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[80] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[81] Ana Nunes Alonso, João Abreu, David Nunes, André Vieira, Luiz Santos, Tércio Soares, and José Pereira. Building a polyglot data access layer for a low-code application development platform: (experience report). In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 95–103. Springer, 2020.

[82] OpenAI. Introducing ChatGPT — openai.com. https://openai.com/blog/chatgpt, 2022.

[83] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[84] PyTPCC, 2024. URL https://github.com/apavlo/py-tpcc.

[85] alronz, 2024. URL https://github.com/alronz/Factors-Influencing-NoSQL-Adoption.

[86] Cypher Queries 1, 2024. URL https://neo4j.com/blog/sql-vs-cypher-query-languages.

[87] Cypher Queries 2, 2024. URL https://neo4j.com/docs/getting-started/cypher-intro/cypher-sql.

[88] MySQL. https://www.mysql.com/, 2024.

[89] PostgreSQL. https://www.postgresql.org/, 2024.

[90] MongoDB Queries, 2024. URL https://www.mongodb.com/docs/manual/reference/command/#user-commands.

[91] Codeium, 2024. URL https://codeium.com/.

[92] Total Priced KB. 3. tpc-c–the standard benchmark for online transaction processsing (oltp). *HISTORY*, 1(253):45–540, 1992.

[93] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013. URL http://www.vldb.org/pvldb/vol7/p277-difallah.pdf.

[94] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[95] OVHcloud, 2024. URL https://www.ovhcloud.com/.

[96] Hetzner, 2024. URL https://www.hetzner.com/.

[97] Pavel Duchovny. Generating mql shell commands using openai and new mongosh shell. https://www.mongodb.com/developer/products/mongodb/generate-mql-with-mongosh-and-openai/, 2023.

[98] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.

[99] Zilin Chen, Liam O'Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. The cogent case for property-based testing. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, pages 1–7, 2017.

[100] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data*, 18(6):1–32, 2024.

This work is co-financed by Component 5 - Capitalization and Business Innovation, integrated in the Resilience Dimension of the Recovery and Resilience Plan within the scope of the Recovery and Resilience Mechanism (MRR) of the European Union (EU), framed in the Next Generation EU, for the period 2021 - 2026, within project ATE, with reference 56.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. DOI 10.54499/LA/P/0063/2020