NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

PARTIAL REPLICATION IN GEO-DISTRIBUTED DATABASES

ANDRÉ DOS REIS MARTINS RIJO

Master/BSc in Computer Science and Engineering

Thesis Plan
DOCTORATE IN COMPUTER SCIENCE

NOVA University Lisbon

Draft: *June 13, 2025*

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

PARTIAL REPLICATION IN GEO-DISTRIBUTED DATABASES

ANDRÉ DOS REIS MARTINS RIJO

Master/BSc in Computer Science and Engineering

Adviser: Nuno Manuel Ribeiro Preguiça
NOVA University Lisbon

Co-adviser: Carla Ferreira
NOVA University Lisbon

Thesis Plan
DOCTORATE IN COMPUTER SCIENCE

NOVA University Lisbon
Draft: *June 13, 2025*

Abstract

Regardless of the language in which the dissertation is written, a summary is required in the same language as the main text and another summary in another language. It is assumed that the two languages in question are Portuguese and English.

The abstracts should appear first in the language of the main text and then in the other language. For example, if the dissertation is written in Portuguese the abstract in Portuguese will appear first, then the abstract in English, followed by the main text in Portuguese. If the dissertation is written in English, the abstract in English will appear first, then the abstract in Portuguese, followed by the main text in English.

In the LaTeX version, the NOVAthesis template will automatically order the two abstracts taking into account the language of the main text. You may change this behaviour by adding

```
\abstractorder(<MAIN_LANG>):={<LANG_1>,...,<LANG_N>}
```

to the customization area in the document preamble, e.g.,

```
\abstractorder(de):={de,en,it}
```

The abstracts should not exceed one page and, in a generic way, should answer the following questions (it is essential to adapt to the usual practices of your scientific area):

1. What is the problem?
2. Why is this problem interesting/challenging?
3. What is the proposed approach/solution?
4. What results (implications/consequences) from the solution?

Keywords: Keyword 1, Keyword 2, Keyword 3, Keyword 4, Keyword 5, Keyword 6, Keyword 7, Keyword 8, Keyword 9

Resumo

Independentemente da língua em que a dissertação esteja redigida, é necessário um resumo na mesma língua do texto principal e outro resumo noutra língua. Pressupõe-se que as duas línguas em questão sejam o português e o inglês.

Os resumos devem aparecer primeiro na língua do texto principal e depois na outra língua. Por exemplo, se a dissertação for redigida em português, o resumo em português aparecerá primeiro, seguido do resumo em inglês (*abstract*), seguido do texto principal em português. Se a dissertação for redigida em inglês, o resumo em inglês (*abstract* aparecerá primeiro, seguido do resumo em português, seguido do texto principal em inglês.

Na versão LaTeX o template NOVAthesis irá ordenar automaticamente os dois resumos tendo em consideração a língua do texto principal. É possível alterar este comportamento adicionando

```
\abstractorder(<MAIN_LANG>):={<LANG_1>,...,<LANG_N>}
```

à zona de customização no preâmbulo do documento, e.g.,

```
\abstractorder(de):={de,en,it}
```

Os resumos não devem ultrapassar uma página e, de forma genérica, devem responder às seguintes questões (é essencial adaptá-los às práticas habituais da sua área científica):

1. Qual é o problema?
2. Porque é que é um problema interessante/desafiante?
3. Qual é a proposta de abordagem/solução?
4. Quais são as consequências/resultados da solução proposta?

Palavras-chave: Palavra-chave 1, Palavra-chave 2, Palavra-chave 3, Palavra-chave 4

Contents

List of Figures

Glossary

This document is incomplete. The external file associated with the glossary 'main' (which should be called `template.gls`) hasn't been created.

Check the contents of the file `template.glo`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

You may need to rerun LaTeX. If you already have, it may be that TeX's shell escape doesn't allow you to run makeindex. Check the transcript file `template.log`. If the shell escape is disabled, try one of the following:

- Run the external (Lua) application:

  ```
  makeglossaries-lite "template"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "template"
  ```

Then rerun LaTeX on this document.
This message will be removed once the problem has been fixed.

Acronyms

This document is incomplete. The external file associated with the glossary 'acronym' (which should be called `template.acr`) hasn't been created.

Check the contents of the file `template.acn`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

You may need to rerun LaTeX. If you already have, it may be that TeX's shell escape doesn't allow you to run makeindex. Check the transcript file `template.log`. If the shell escape is disabled, try one of the following:

- Run the external (Lua) application:

  ```
  makeglossaries-lite "template"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "template"
  ```

Then rerun LaTeX on this document.
This message will be removed once the problem has been fixed.

Symbols

This document is incomplete. The external file associated with the glossary 'symbols' (which should be called `template.sls`) hasn't been created.

Check the contents of the file `template.slo`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

You may need to rerun LaTeX. If you already have, it may be that TeX's shell escape doesn't allow you to run makeindex. Check the transcript file `template.log`. If the shell escape is disabled, try one of the following:

- Run the external (Lua) application:

  ```
  makeglossaries-lite "template"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "template"
  ```

Then rerun LaTeX on this document.
This message will be removed once the problem has been fixed.

1 Introduction

In the current days, new services and applications appear online everyday. Billions of devices are connected to the Internet, interacting between themselves in different forms. Applications such as Spotify, Netflix, Instagram, Amazon, Youtube, to mention but a few, operate in a scale unimaginable a couple of years ago, with dozens of millions of users active everyday. Supporting world-wide scale applications has several implications. First, users expect services to be fast - studies show that even a slight increase to reply time can lead to considerable profit losses [31, 60, 38, 73]. Second, services are expected to be available 24/7, despite failures happening frequently. Third, services must work correctly at all times - data should always be consistent. To cope with those requirements, world-wide services deploy multiple data centers (each with many servers) spread around the world.

As the scale of the services increases, so does the amount of data. Traditionally, data is replicated everywhere, thus as the amount of data centers increases, so does the number of copies of each data. This increases storage, networking and processing costs, as data must be kept up-to-date everywhere. Depending on the system consistency guarantees, this may also slow down user's operations, due to the time required for faraway data centers to confirm operations [42, 46, 64].

1.1 Motivation and challenges

A key insight is that in many large scale services, specially if world-wide, data does not need to be replicated everywhere. For instance, consider an e-commerce application with world-wide coverage. The information regarding Portuguese customers, as well as stock of products in Portugal, does not need to be replicated everywhere - it is unlikely to be relevant outside of Europe. Thus, it can be replicated in European data centers and, e.g., North American ones for fault tolerance purposes. With this, we reduce the following costs: storage (each data center holds less data), network (updates of an object are only sent to a subset of the data centers) and processing (less updates in total for each data center to process). The method of replication in which each data center only replicates a

subset of the system's data is known as partial replication. Partial replication is not a holy grail though - multiple issues arise when data is not replicated everywhere.

First, under partial replication, maintaining application properties (invariants) valid at all times is rather challenging. E.g., consider an invariant regarding sales of tickets for a concert across multiple sale spots - we may want to ensure we do not sell more tickets than the capacity of the venue. The data regarding those sales may be spread across different data centers. Different levels of consistency or coordination may be required to ensure correctness, depending on the criticality of an operation.

Honestly, I do not know where (in this Motivation subsection) should this part of invariants be. Suggestions are welcome.

Second, the replication algorithm running in each data center must know, for each object it replicates, which other data centers also replicate said object. This can be difficult to dealt with, as the partitioning may change over time and the distribution of objects may not be uniform across data centers - that is, each data center may replicate different subsets of data, potentially with overlaps. E.g., DC1 replicates object 'A' and 'B', while DC2 replicates 'A' but not 'B' and DC3 replicates 'B' but not 'A'.

Furthermore, supporting transactions under partial replication is challenging. On one hand, a transaction may span objects that are all present in the origin data center, but whose objects are spread across different places in other data centers, as in the example above [29]. One solution would be to send the entire transaction to all data centers with at least one of those objects, but this would be inefficient. It is thus necessary to design an algorithm capable of analysing a transaction and sending only the necessary information for each server, while still ensuring data is kept consistent everywhere. A proper algorithm will both reduce the amount of data sent and processing required in the receiving end.

Another problem arises regarding transactions. Even with an optimal partitioning scheme, which is a difficult problem on its own, sometimes a client may issue transactions to a data center where some operations are for objects not replicated locally [10]. This poses many difficulties. One could consider splitting a transaction, sending each part to one data center. But this may require coordination, increasing latency. It is also difficult to ensure consistency of data read in different places without further increasing latency. Another solution would be to still execute the transaction as a whole, but have the data center starting the transaction coordinate/fetch the necessary information in some form. However, this still leads to the aforementioned problems. In fact, if this kind of transactions are common, it may even be better to fall back to full replication. Thus, a better alternative consists in creating mechanisms to reduce the frequency of cross data centers transactions. Regardless, any system employing partial replication must consider this issue.

The issue above is further exacerbated when we consider complex recurring queries, specially analytic or OLAP queries [45, 41, 75]. In analytic queries, often large amounts of data need to be processed [72, 88]. For instance, in an e-commerce system, one may want to query the "top 10 most sold products world-wide". This kind of queries are essential to make well-informed business decisions [72, 8]. If a product suddenly becomes very popular, supply chains may need to be reinforced quickly to maintain stock, and it may be wise to make the product easier to find in the application. However, to compute this

kind of information, a large sum of data needs to be processed - potentially sales of all products. Yet the query must be answered quickly and correctly - if not, the relevant business decision may happen too late, leading to profit losses or negative impact on customer satisfaction. This problem is already quite interesting and challenging even under full replication, as it is difficult to process large sums of data (long-executing query) while keeping results consistent and without hindering ongoing queries and updates [88]. Partial replication complicates this problem ten-fold, as now the necessary data is spread across many different and faraway data centers. Proper mechanisms are essential, as coordinating long-executing queries across faraway data centers is not viable [72].

In short, bringing partial replication to large scale, world-wide distributed applications is essential to reduce operating costs of systems and keep them scalable. However, partial replication arises new issues due to data not being available everywhere and further exacerbates already existing problems and challenges.

1.2 Proposed solution

In our work, we plan to tackle the aforementioned issues. In general terms, we intend to design algorithms optimized for partial replication which address the challenges previously mentioned. We also aim to implement a geo-distributed database with partial replication that shows our vision and implements our algorithms, which will be used both as a proof-of-concept and to evaluate our algorithms.

Starting from the end, we aim to provide materialized views [41] as part of our answer to recurring, analytic queries regarding large amounts of data. Materialized views will hold the necessary data to answer the query efficiently. For example, one can define a view that contains the list of the most sold products. Our provision of materialized views will be novel, as they will be operating under partial replication - namely, our views will be able to relate/summarize data that may be partitioned across multiple data centers, without any single entity that has all the data necessary to fully compute the view. This implies developing algorithms which keep the view up-to-date and consistent, despite partial replication of the necessary data. Efficiency of view updating is also essential, as materialized views are often considered expensive to maintain.

The replication and transactional algorithms we will need to design and implement will also be both novel and challenging. Not only they will have to coupe with partial replication (which has the challenges mentioned previously), an even more interesting challenge comes in how to support view updating without all the necessary data being present in one data center. This implies that our replication algorithm will have to ensure views are kept up-to-date whenever a relevant object is updated, even if many replicas of the view do not replicate said object. Our transactional algorithm will also have to ensure the views stay consistent with the objects they refer to, in order to avoid animalities which affect user experience and application development. Thus, our algorithms need not only to cope with partial replication, but also with the challenge of providing views

I think I might be giving too many details, and also some repetition. But it is difficult - replication, transaction and views are all too connected.

Should I maybe say "some of the aforementioned issues"?

with a global scope under the scenario of partial replication. Furthermore, we also have to tackle common issues of materialized views, namely storage and maintenance costs by, e.g., incorporating non-uniform replication in our replication algorithm and using appropriate data types to hold the necessary view data.

Finally, in order to ensure latency optimality of transactions, weak consistency [21, 80, 59, 31, 77] will be used. However, it is hard to program on top of weak consistency systems, as many anomalies are exposed to the programmer [21]. Thus, in our prototype, we will deploy causal consistency [5, 19] and extend the model to our views. Furthermore, we will reinforce our consistency guarantees with invariants that will be maintained by specialized data types and mechanisms, as well as offering different levels of consistency - both of which must support partial replication and views. Finally, our system will support both SQL [26] and a key-value interface, which must be intercompatible. This by itself is already quite challenging, as SQL is very expressive. All these mechanisms are essential to ensure our algorithms and mechanisms are as practical and accessible as possible to developers.

Note: I did not motivate the need for SQL/key-value interface in the previous subsection.

1.2.1 Expected contributions

Is an expected contributions subsection necessary? In a way it is already described what will be our main contributions, and Chapter 3 will detail this more anyway

Given the aforementioned challenges and proposed solution, we expect the main contributions of the work to be:

- Design and implementation of algorithms for efficient geo distributed replication, combining partial and non-uniform replication to reduce network, storage and processing costs;
- Design and implementation of a transaction algorithm supporting multiple consistency levels;
- A solution for analytic queries spanning high amounts of data though the usage of materialized views. Views may span data partitioned across many servers and data centers. Views must also stay consistent and up-to-date despite weak consistency;
- Provision of useful correctness guarantees to application developers, both through employment of useful consistency models and invariants;
- A database implementation of all algorithms developed in this work. Our database will provide varied and rich data types well suited for geo distributed applications.
- A rich database interface with support for both SQL and key-value style access, extended with useful operations.

1.3 Document organization

The rest of this document is organized as follows. First, Chapter 2 covers and analyses the state-of-the-art in relevant fields. Afterwards, Chapter 3 discusses in detail the challenges

and goals of this work, covering both challenges/goals already addressed and future work. Finally, Chapter 4 outlines the plan for the remaining time of this PhD, namely the timeline for each of the remaining goals.

2

Research Context

Is an Introduction like this okay? Or is just mentioning what will be discussed in this chapter enough?

In this research work we propose to provide a geo-replicated, weakly consistent database named PotionDB which provides reliable access to complex analytical queries in near instant time. Achieving this goal requires combining and leveraging on multiple concepts, for which multiple solutions exist and thus it is necessary to make informed choices and leverage on existing solutions/concepts. This chapter discusses such concepts.

The chapter is organized as follows. Section 2.1 describes the CAP theorem, laying the foundations to the diversity of solutions for data storing. Section 2.2 discusses how storage systems handle data consistency. Section 2.3 introduces multiple concepts related with data replication, raging from synchronization to data representation and distribution. Section 2.4 focuses on databases and efficient query provisioning, closing up with a discussion of existing solutions that apply the concepts introduced in this chapter. We also discuss relevant choices we have made in our system regarding these concepts.

Note: I am not sure if I should had talked about Antidote/Cure in high detail or not. I talked a bit about Cure on the Existing Systems subsection, is that enough, or does Antidote/Cure deserve a section of its own?

2.1 CAP theorem

In an ideal world, distributed systems would all provide strongly consistent data, be always available and keep operating even in the presence of failures and network partitions. However, the CAP theorem [34] states that it is only possible to, at most, provide two out of the three following properties: (i) Strong **C**onsistency - all replicas of a system must provide the same view of the data at any given time; (ii) **A**vailability - the system is capable of providing a reply to every client's request at any point in time; (iii) **P**artition-tolerance - the system keeps operating and evolving state correctly even in the presence of message loss and network partitions.

Typically, since network partitions are common on systems distributed across multiple regions [21], the choice tends to be between providing Strong Consistency or Availability. It is noteworthy that one does not need to choose "all or nothing" [21]. Systems providing **C**onsistency and **P**artition-tolerance (CP), can still be available in the presence of some faults, but will eventually stop to evolve state if too many nodes fail or too many messages are lost [34]. On the other hand, systems providing **A**vailability and **P**artition-tolerance

Do I need to make more of a connection between the topics and PotionDB? E.g. motivate what we will be doing, how our solution is different/better, etc.?

can still provide some consistency guarantees, known as Weak Consistency. However, the replicas will diverge at times and conflicting states may be observed due to the concurrent execution of operations [21].

2.2 Consistency

In the literature, a plethora of consistency models have been specified [84, 33, 29, 21, 59, 31, 48], each one providing different guarantees to the developers in terms of the states that can be observed, as well as different requirements to deploy such models. Despite the variety of models, they can be grouped as either being a form of *Strong Consistency* or *Weak Consistency*.

2.2.1 Strong consistency

In strong consistency models, all clients observe the same state of the system independently of the replica contacted by the client(s). To achieve this, it is necessary to totally order all write operations executed in the system, which requires coordination among the different replicas [84, 29, 42, 46, 64, 7]. Depending on the system configuration, even to execute a single read operation it may be needed to contact a majority of replicas or a master replica [29, 69]. This is necessary in order to ensure the client always reads the latest correct state.

The strong requirements of coordination imply that strongly consistent systems, in order to tolerate network partitions, may have to come to an halt (unavailable) when enough replicas can not be contacted to execute an operation, in order to prevent state divergence or incorrect values being returned [39, 5, 7, 31]. Furthermore, the costs of synchronization can be prohibitively high for large scale systems, specially if replicated across the globe, as latency for operations quickly racks up [42, 46, 64]. Also, due to the total order requirement, it is often difficult to scale to a high number of clients and maintain high throughput under contention [47, 31]. On the other hand, strongly consistent systems tend to be easier to develop applications on than weaker consistency, as they are easier to reason about and do not expose consistency anomalies [29, 39, 69, 7, 31].

A few models implementing strong consistency have been proposed in the literature [84, 33], such as: serializability, linearizability and snapshot isolation. Recent research focuses on lowering the number of round trips (RTTs) needed for transaction execution, as well as leveraging on batching, data locality and other similar techniques, in order to reduce commit latency and improve throughput [39, 42, 64, 69, 27].

2.2.2 Weak consistency

Weak consistency models are considerably more relaxed in terms of requirements compared to strong consistency [21]. As such, they also provide weaker guarantees, making them harder to reason about and build applications on [47, 39, 69, 59, 5, 7, 31]. Indeed, concurrent updates on the same objects may lead to undesired results or values. Replicas'

state may diverge, with read anomalies and inconsistent states being observed [47, 64, 69, 59, 5, 31]. For example, in a group communication system, under some consistency models, user A may see user B replying to user C's message before he even sees user C's message. This can be very confusing to the users. However, the weaker requirements of weakly consistent systems gives potential to many advantages compared to strong consistency: better fault tolerance (operations keep executing even under network partitions), lower latency and higher scalability in both throughput and number of clients [69, 59, 5, 31, 47].

One of the most basic consistency models is eventual consistency [21]. In short, eventual consistency provides only one guarantee: when updates stop occurring, eventually the state of the replicas will converge. However, several issues are left in the open - e.g., what happens when the same object is concurrently updated? Which states can be observed on reads? How do the states evolve as updates are being applied? Thus, many other weak consistency models have been studied in the literature [21, 80, 59, 31, 48, 77, 28], e.g. causal consistency, causal+, monotonic reads, read committed, per record sequential, PSI, etc.

Causal consistency [59, 60, 7, 5] is of particular interest as it provides useful guarantees. In a general form, causal consistency ensures that operations casually related [50], i.e. in which one happens before the other, are seen according to said order by every client. This means that, e.g., a read that happens after a write must reflect the effects of said write. With causal consistency, the aforementioned example of group communication system would no longer be possible - user A would always see user B's reply after seeing user C's message, as they are causally related [50, 77].

Any system that provides causal consistency on the level of a single object only? I couldn't find any other than... a CRDT itself. (I do not think PNUTS counts as single-object causality, as some authors call it "same-key sequential consistency")

Causality can be provided either in the context of a single object (i.e., causality between different objects is not ensured) or between multiple objects [21]. The former is tougher/costly to maintain, as an order needs to be maintained between potentially all objects, but more useful. For example, in a online shopping system which allows users to track when a product is restocked, we want for the notifications sent to the users to be causally related with the restocking action, as we do not want users to receive the notification before they can see the product in stock. For this use case (as well as the group communication example, among others), causality between objects is very useful and eases application development.

Causal+ consistency [59, 7, 5] extends causal consistency by ensuring replicas do not diverge forever, through techniques that solve concurrency conflicts deterministically in all replicas. By definition, causal consistency does not ensure replicas converge [59, 7].

Monotonic reads [80, 21] gives one simple to understand guarantee: consecutive reads on the same object must always return the same value or a more recent value. This means that after a value is read, it is not possible to read again an older value. It is however possible to read stale data, or keep reading the same value despite more recent values existing. In distributed systems, it is important to define if this guarantee is given at a single server level or across servers, as the latter may imply having to wait for data to arrive and be processed if the client issues reads in different servers.

Snapshot Isolation (SI) is an interesting model. The intuition behind SI is that each

transaction sees a consistent view of the database [33] - a "snapshot", where all objects are at the same version and the database is in a consistent state. Under SI, the transactions in a replica are totally ordered, thus preventing concurrency conflicts when updating the same objects. This implies coordination is required to commit a transaction. In SI, reads on a transaction are executed on a snapshot, never blocking. Writes are applied on a higher version when the transaction commits, if no other transaction that committed in the meantime modified the same object(s). However, some anomalies can still occur under SI. E.g., consider objects 'c' and 'd', and transactions 'A' and 'B'. Consider now that both transactions read 'c' with value of 5. If transaction 'B' updates 'c' to 10, and transaction 'A' reads 'c' and copies the value of 'c' to 'd', both transactions can commit. However, it is possible for 'B' to be ordered before 'A' and yet, end up with 'c' = 10 and 'd' = 5. This would not be possible under serialization.

Parallel Snapshot Isolation (PSI) extends SI by allowing transactions issued in different replicas to commit in different orders [77]. PSI is useful for weak consistency systems, as they can provide strong guarantees when clients access only one server and, e.g., causality between updates on different replicas, thus still allowing high throughput and low latency in some cases. However, under PSI, coordination may still be needed for some transactions, as PSI guarantees there are no write conflicts.

Sovran et. al. [77] define PSI+cset as an extension of PSI. This extension states that if all operations of a given data type are commutative between themselves (e.g., a counter with only increment and decrement), then updates on objects of said data type never conflict.

In PotionDB we provide PSI+cset, with causal+consistency (hereafter causal consistency) for transactions executed in different servers. We provide causality even if the client contacts a different server. Our causality guarantees are cross object. The views in our system are directly co-related to data of other objects, thus when a client observes a change in one of such objects, it must also observe the corresponding change in the views. Causality between objects and views is very challenging in PotionDB's scenario, as data is partially replicated. Thus, views can be related to objects of different partitions, without a single server having said view and all of its objects. Causality must still be ensured despite this difficulty. In PotionDB synchronization is never required to commit a transaction, as all our objects only support commutative operations - thus there are no write conflicts.

To accommodate use cases in which higher throughput is desirable and causality is not needed for the read, we also provide monotonic reads as long as the client is sticky to one server. Note that even with monotonic reads, in PotionDB the client only observes states generated by committed transactions.

2.2.3 CRDTs and other conflict resolution techniques

Both with eventual and causal consistency, concurrent updates on the same objects can occur and lead to conflicts [21]. For instance, if an element is concurrently added and removed from a set, it is not clear what should be the final state. Concurrency conflicts

I do criticize this hybrid approach in the geo replication section.

Is this paragraph ok? Or should I say something like "PotionDB will provide (...)"?

Is "We provide causality even if the client contacts a different server" clear? My intention: when a client stops contacting its server and contacts a different one, causality is still kept, as he/she will ask for a snapshot more recent than the last one he/she observed.

Should this last phrase be in this subsection,

need to be dealt with, in order for all replicas' state to converge. Solutions for handling conflicts can range from preventing them through coordination (as in PSI without csets), totally ordering operations following some criteria (e.g., using logical clocks and replica IDs), letting the user/application deal with the conflicting state (e.g., in a register with concurrent writes, use application logic to decide which value to stay) or forcing all operations to be commutative (if operations commute then the result will be the same, independently of the order by which they are applied).

In PotionDB we make usage of CRDTs [74] to represent our datatypes. CRDTs, or **C**onflict-Free **R**eplicated **D**ata **T**ypes, are data types designed to be used in large-scale distributed systems offering weak consistency. Concurrency conflicts are solved by the CRDT itself, by designing operations to be commutative. Updates and reads can execute locally without synchronization, with all replicas of a CRDT eventually converging. Updates get propagated asynchronously. This allows for high availability and low latency on accessing CRDTs, as it is even possible to cache CRDTs on the client-side [58, 87, 25]. Multiple policies for handling conflicts are available for the same datatypes, allowing CRDTs to more easily adapt to the needs of each application [70, 74].

Alternative solutions based on commutativity have been studied in the literature. One such solution is RADTs (**R**eplicated **A**bstract **D**ata **T**ypes) [71], which are similar to CRDTs but with more limited conflict resolution policies. Another solution is OT, **O**perational **T**ransformation [79, 66]. The intuition behind OT is that conflicting operations can, when arriving on a replica, be "transformed" to be commutative and thus not conflict. However, it is usually considered harder and more error-prone to design said transformations instead of, as in CRDTs, designing operations to be commutative from the start [66, 74]. We believe CRDTs' ease of use and flexibility with solving conflicts justify its usage over OT-based solutions in PotionDB.

2.3 Replication

In distributed systems, replication is essential to ensure data is available at multiple sites. Depending on the system, replicating data may lead to better fault-tolerance, lower operation latency and higher scalability or throughput [5, 47, 60, 52]. However, it also implies higher storage, network and computational costs [10].

It is thus essential to tailor replication specifically to the system's intended use-case. Factors like consistency model, fault model, read/write ratio, access patterns, types of operation and geo-distribution of the servers affect the choice on how to efficiently replicate data. As will be seen in this section, in PotionDB we combine ideas from multiple replication concepts to provide a novel replication model tailored for PotionDB's use case.

2.3.1 Synchronous and asynchronous replication

Are the definitions here of synchronous and asynchronous replication clear

In synchronous replication, an update is only considered completed after it has been replicated to other replicas and acknowledged by a subset (often majority) of them [62]. This is usually applied by strongly consistent systems in order to keep the replicas up-to-date and avoid concurrency conflicts [31, 29, 47, 64]. These systems usually employ protocols such as Paxos [49] or Total Order Broadcast [83] to totally order the operations, or a "master-slave" model to sort operations in one place [10, 46, 64, 69]. These mechanisms, together with synchronous replication, ensure no operations are lost and clients always read the latest, correct state [29, 46, 39, 62].

In asynchronous replication, an update may be confirmed before being sent to other replicas [62]. This approach is ideal for weakly consistent systems, enabling swift confirmation of operations to clients [59, 5, 39]. Replication occurs in the background, possibly deferred to promote batching with other operations for efficiency [31, 5].

Synchronous replication provides strong guarantees regarding state evolution of systems, ensuring updates are not lost and data is always consistent. However, having to wait for the confirmation of other replicas limits fault-tolerance [29, 42, 46, 39] and increases operation latency when replicas are faraway, making it unusable for geo-distributed systems or latency-sensitive operations [69, 47, 62, 77]. Asynchronous replication enables lower latency and higher throughput, as operations are confirmed locally without waiting for other replicas [5, 47, 59]. It also has lighter network requirements. However, state diversion and concurrency conflicts may arise from lack of replica coordination and must be addressed [31, 46, 47, 59]. While asynchronous replication may lead to more fault-tolerant systems [31, 48, 5], namely in the presence of network partitions, replica failure may result in data loss, as some confirmed updates may not have been yet replicated.

Is it worth it to make a small section to present more information on state machine replication and paxos?

Either here or in Research Context I should mention we make usage of async replication and explain why (avoid latency increase from geo-replication; better fault tolerance; sync not needed for our use-case)

2.3.2 Operation and state based replication

Replicas can be kept up-to-date in different forms. One important decision is on how to propagate changes to objects' state. In the literature, solutions for propagating changes are usually grouped as either *state-based* or *operation-based* [74].

State-based [74]. Updates are propagated by sending the entire object state to other replicas. If updates happen concurrently in multiple replicas, the coming state may have to be merged with the existing state, so that the effects of operations in the existing state are not lost. Merges and state propagation must be done in such a way that it is guaranteed that, eventually, all replicas converge to the same state. Extra care must be taken when applying consistency models stronger than eventual consistency - e.g., in causal consistency, merges must preserve causality [21].

Operation-based [74]. Updates are propagated by directly sending operations or their effects to other replicas. Receiving replicas must then apply the operations or effects on their own objects. Depending on the consistency model, a delivery and/or appliance order may

Should this section be more CRDT-focused (as in, the case of CRDTs in particular) or generic (the generic concept of state-based/op-based replication)

be required - e.g., causal or total order. Care must also be taken with message duplication. Moreover, all relevant operations must be delivered everywhere and concurrency conflicts must be handled properly to prevent states from diverging forever.

CRDTs [74]. CRDTs are classified based on their type of propagation: state-based CRDTs and operation-based CRDTs (op-based CRDT), respectively.

In state-based CRDTs, ensuring eventual state convergence requires that the ordered states of an object form a monotonic join semi-lattice, and the merge operation must compute the least upper bound [74, 16]. The intuition is that, for any two states, a merged state can be calculated that is not older than either of the merging states, and reflects the effect of all operations applied on both states. Since states can be merged in any order, eventual delivery of states to every replica is enough. In practice, not every single state must be delivered, but rather, enough recent states that ensure that, when updates stop happening, all replicas converge into the latest, correct state.

In op-based CRDTs, every update must be delivered to all replicas. If all operations are commutative and idempotent, eventual delivery suffices. Often, only concurrent operations are commutative and thus casual delivery is required from the network [21].

Shortcomings and optimizations. Both approaches have shortcomings. Operation propagation can overwhelm a system with numerous small messages when updates are very frequent, while state propagation can be too costly with big states, resulting in large messages even for minor state changes. Both approaches can be optimized. For op-based, operations can be batched in a single message and applied in a row. For states, it is possible to calculate differences between states (delta) and send instead only the differences. This solution has been studied in the literature [6, 57], with CRDTs based on deltas being known as delta-CRDTs. Delta-CRDTs present considerable reductions on message size compared to state-based CRDTs, but also have complex specifications [6]. Also, keeping replicas correctly up-to-date is challenging, specially with causal consistency [6].

2.3.3 Partial replication

Full replication is popular in distributed storage systems [10, 17]. It enables high fault tolerance by replicating data everywhere, ensuring its safety (and, in some consistency models, its availability) even if several servers crash or are unreachable due to network partitions [5, 62, 52]. It also allows data to be close to users anywhere in the world.

However, this approach has limited scalability, [10, 60]: as server count grows, storage, network and processing costs rise as each server stores all data and receives and processes every update. For large-scale systems these costs may become prohibitive. The literature proposes two different mechanisms, sometimes combined, to alleviate replication costs:

Internal partitioning (sharding). Sharding splits the dataset within each server or data center, enabling the system to scale with multi-core CPUs or number of servers, respectively.

The latter is often used to handle datasets too large to fit in a single machine's storage [72]. It also reduces the growing replication cost associated with high server counts, through it introduces overhead for accessing data across partitions [31, 59, 46, 62]. Furthermore, transaction execution may require synchronization between the shards, but this may be acceptable as latency inside a data center is low [59, 46, 64, 78]. Nonetheless, this model still replicates all data across all data centers, which may be unnecessary and waste storage and network resources, as well as create latency issues.

External partitioning. With external partitioning, each data center may replicate a different subset of data. This allows to further reduce storage, network and processing costs as less data is stored and less updates need to be sent and processed in each data center [29, 10, 17]. External partitioning is often referred as partial replication.

Partial replication is thus of interest for large-scale services to keep replication costs bearable as the number of data centers increases. It is especially useful for systems distributed all around the world with data locality [10, 69, 64, 42]. E.g., in an international e-commerce application, European customers, sales and stocks data is likely not relevant in Asia. With partial replication, such data could be replicated only in European data centers and optionally in another region for fault tolerance. Additionally, partial replication enables efficient synchronization (e.g., to ensure strong consistency), as transactions regarding European data only need to coordinate with data centers holding said data, instead of all data centers [29, 10, 69, 42, 39], enabling lower latency and higher throughput.

However, external partitioning needs to be carefully done, factoring in application context. Since each data center does not have the full dataset, it is essential for the majority of transactions to access only data which is within a single data center. Otherwise, transactions may need to contact or synchronize with multiple data centers [29, 10, 36, 69], even under weak consistency, which severely hindering throughput and often leading to high latency, resulting in a degraded user experience [60, 46, 22]. Fine-turning the partition scheme is therefore crucial, with research being done to enable systems to adapt to access patterns changes by automatically migrating data between data centers [69, 64, 1, 52, 10]. If there is no data locality or some other precise partitioning criteria, full replication may be more adequate, specially under weak consistency.

Do I need some concluding note here/reference to PotionDB? Is this too much information?

2.3.4 Geo replication

Many large-scale Internet services have users spread across the globe, expecting low latency access no matter their location. As such, these services require data centers spread worldwide (geo distributed). We define a system as geo replicated if it replicates data across faraway data centers. Geo replication is challenging due to high latency between faraway data centers, alongside limited bandwidth [46, 60, 36, 69]. Availability together with strong consistency is specifically difficult, as previously discussed in Section 2.1.

A related concept is wide-area replication. In wide-area replication, multiple data centers are still employed, but with limited distance between all data centers (e.g., all located in the United States). Some systems in the literature were designed with wide-area replication in mind [29, 59, 12]. In this scenario however, coordination between data centers may be acceptable, as latency is still low enough [29, 3] (below 100ms[1]). In geo-replication, this coordination may become prohibitively expensive, as latencies between faraway data-centers are in the orders of hundreds of milliseconds, which can easily degrade the user experience and drive users away users [46, 60, 36, 3, 22].

Given the reasons above, solutions designed with geo-distribution in mind are required. Solutions combining strong consistency and geo-replication exist [46, 69, 37], but they usually have very high latency, limited availability and sometimes low throughput. Some systems try to circumvent those shortcomings [46, 36, 69, 77, 42, 64], e.g. by leveraging on partial replication, but the problem persists when multiple data centers need to be accessed. Additionally, it often comes at the cost of fault-tolerance, as partial replicas of the same data tend to be geographically close [42, 64, 69, 82]. Other systems offer strong consistency for transactions within a data center and weak consistency across data centers [59, 60, 77]. However, this requires applications to deal with issues inherit to weak consistency, as updates are replicated under weak consistency. eybog. Finally, solutions offering weak consistency are attractive for geo replication, as they tend to cope better with high latency, partial replication and scalability [60, 5, 77, 31]. However, systems offering only eventual consistency (e.g. Dynamo [31]) are hard for application programmers to develop on, due to very weak consistency guarantees. Thus, research has been done to bring causal consistency to wide and geo distributed systems [59, 60, 19, 5, 77, 7], with causality across objects, specially under partial replication, being particularly challenging. Weak consistency systems usually provide low latency as they do not require coordination among data centers, but even with causality, applications still need to deal with some consistency animalities [39, 62].

2.3.5 Non-uniform replication

Partial replication already reduces network and storage costs by reducing the number of replicas of an object. However, it does not help with the case of large objects or popular objects with very frequent updates.

Consider a worldwide e-commerce application, with hundreds of thousands of products (e.g. Amazon). A naive implementation of a ´´top 10 most sold products" leaderboard would keep a counter of total sales per product. This would imply that whenever a product is sold somewhere, an update to this leaderboard is generated and then propagated and processed everywhere. However, the vast majority of the updates do not change the visible state (the top 10), as updates to products outside the top do not affect leaderboard reads.

[1] For instance: Amazon's AWS servers in the United States have below 75ms latency among themselves, while the European's servers have below 60ms [3].

Non-uniform replication [23] is useful for such situations. With non-uniform replication, only updates that change the visible state of objects are propagated. Updates that may be relevant later (e.g., if we remove a product, another one may rise to the top) are locally stored and only partially replicated for fault tolerance purposes, while updates that will never affect the visible state are immediately discarded (e.g., in a max register, adding a value below the max is irrelevant). Thus, non-uniform replication reduces storage, network and processing overheads of large objects, while ensuring read correctness.

Cabrita et. al. [23, 24] propose some non-uniform CRDTs, including designs for Top-K (leaderboard), Top-Sum (leaderboard but supports increments to entries), histograms and filtered set. Another related concept is computational CRDTs [63]. In computational CRDTs, the client is not interested in the complete state of an object, but rather the result of some computation over the state. E.g., in an Average CRDT, only the average is relevant for the client, and not each added value. As such, computational CRDTs keep only the strictly necessary information to calculate the result (in this case, the sum of all adds and number of additions), instead of all inserted values, thus reducing storage usage.

2.3.6 Replication in PotionDB

PotionDB is a geo-replicated database that supports both partial and full asynchronous replication. System administrators can decide, for each group of objects, which ones are fully or partially replicated, and where. Replication is asynchronous and operation-based. PotionDB also employs sharding inside each server to leverage on multi-core CPUs.

Finally, PotionDB supports multiple non-uniform and computational CRDTs, including some new ones we designed (e.g., max/min CRDT) and extensions (e.g., adding decrement support to TopSum CRDT). We discuss their usefulness in PotionDB in Chapter 3.

Update this link to the correct section

2.4 Data access

Online services usually require storage solutions to hold their data. Databases are commonly used to store and manage an application's data, with different types of databases offering different guarantees (consistency, durability, etc.) and interfaces for data access.

While there are many different kinds of database interfaces, the two most common ones are key-value stores and relational databases. Most alternatives can be defined as extensions, variants or subsets of these.

Key-value stores. Most key-value stores present a simple interface with two basic operations - *get* and *put* [31, 59, 5, 77, 44]. The former allows users to obtain the state of an object, while the latter updates or stores an object. Each object is accessed by providing a key that uniquely identifies it. Some databases only provide opaque objects (i.e., registers), while others may provide richer interfaces by supporting objects such as counters, maps and sets, alongside appropriate operations on those, or other features such as transactions [5,

77, 44]. PotionDB provides a key-value interface, with support for many different kinds of objects, operations, transactions and even for materialized views.

Relational databases. Relational databases provide a so-called relational model [29, 60, 35]. In relational databases, data is organized in tables, with possible relationships between tables and the columns of different tables. E.g., each entry in a table "customers" may refer to one or more entries in the table "addresses". Usually, columns represent the fields of a table, while each row represents one entry (data) in the table. Many systems provide SQL (**S**tructured **Q**uery **L**anguage) or a SQL-like language [26, 29, 35, 2] to define and access data, which allowing for a rich interface for handling data. For instance, with a SQL query, it is possible to obtain directly from the database the average of salaries of users in a given company. Conversely, in key-value stores, usually one would need to read multiple objects, calculate the average and then present the result, or store the pre-calculated result under some key (and then keep it up-to-date manually).

Key-value stores have a simple interface [31], often offering weak consistency and stressing on performance and fault tolerance [31, 59, 5]. Conversely, relational databases' richer interface eases application development [29, 35, 60], but are tougher to design and optimize for, being mostly used by databases providing strong consistency. Some key-value stores provide extra features to facilitate application development by, e.g., providing many object types, extra operations or transactions [5, 77].

2.4.1 Queries

Applications access data in different ways. Some mostly access data directly, e.g., fetch information about a product in an e-commerce system. Others may require summaries, aggregations, sorting of data or even more complex processing.

Online **A**nalytical **P**rocessing, OLAP, is a method consisting on ways to analyse and organize data by executing analytical queries [45, 41, 53, 51], which are often used in the context of marketing, financial reports, business decisions and scientific data analysis. For instance, in an e-commerce system, it is useful to track the trending products in order to, e.g., adjust supply, prices or advertising. This is challenging as very large volumes of data need to be analysed (all sales), with a constant stream of updates (new sales and products) and in a tight time frame (information must be quickly available to enable timely decisions). Common methods to provide analytical queries include views, indexes, windowed data and caches [35, 45, 44, 67, 41, 53, 51]. Operations such as table joins, sums/averages of data, sorts, limits, group by and others are often used by this kind of queries. Current solutions face limitations, often requiring expensive infrastructures, slow query execution or having to accept staleness/coordination/inconsistencies [35, 88, 8, 72, 45, 69].

Is it clear to what is "which" referring to here? My intention: analytical queries

The TPC-H benchmark [30, 45, 89, 90] provides a specific example of a database managing sales data which aims to provide useful information for business decisions. TPC-H's queries are a good example of the kind of queries executed in analytical scenarios.

Nowadays, a key requirement of analytical queries is to provide useful information with queries of quick execution, with constant updates happening in the background.

2.4.2 Speeding up queries - views and indexes

In order to provide applications with simpler and quicker data access, several mechanisms have been studied to complement direct data access [35, 45, 44, 67, 41, 53, 51, 90, 56, 85, 55, 9], including indexes, views, data streaming, key-value stores as caches, etc. We describe here indexes and views, deferring caches and data streaming systems to Section 2.4.4.

Indexes. An index is a record whose goal is to speed up how quickly data is found [85, 2, 56]. In relational databases, an index can be made on a column of a table to provide quick access to data by some criteria other than the primary key. E.g., an index on users' data may provide quick access by phone number. It can also be used to help group data or sort more efficiently. In non-relational databases, indexes can be useful too - e.g., a map can store all users' keys, and each key can be used to directly access the respective user's data. In short, indexes are useful and commonly used to speed up queries and data access.

Views. A view consists in a different way to present data. In the context of databases, it is defined as the result of some query [67]. Views can be used to filter data (e.g., only show users above a certain age), or to accommodate subqueries that may be used by other queries (e.g., in a relational database, a join of two tables). As such, they are useful for application developers, providing data abstraction by letting users define relationships between tables. Views are re-calculated every time they are used, thus they do not use storage space but require processing time when queried. Materialized views, on the other hand, are used with the goal of speeding up queries [35, 44, 67, 89, 41, 40, 51, 90]. Materialized views cache the results of a given query, thus they can provide much quicker access to data [76, 8, 32], avoiding re-execution of certain queries all the time. However, it comes with its downsides - extra disk storage is needed and maintaining the views consistent and automatically updated is challenging [4, 20, 36, 81, 72], thus few systems implement them. View maintenance is a common research topic [35, 20, 4].

2.4.3 Data safety

Data safety or Invariants?

Some applications require certain conditions to hold true at all times (i.e., invariants [14, 15]) to function correctly - e.g., in a banking app, that an account's balance is >= 0. Under strong consistency, this can be ensured by checking the balance before a withdrawal operation - however, this is burdensome and error-prone for developers, as real-world systems often have multiple invariants to keep. Moreover, with weak consistency, checking the condition is not enough due to concurrent operations: e.g., two concurrent withdraws of 10€ may succeed on a 15€ balance, as both withdraws see enough balance available.

As invariants are essential for application correctness, much research has been done towards detecting conflicting operations and automatically holding invariants [15, 11, 28, 86, 54, 43], with a focus on reducing required coordination. However, most solutions still require coordination (i.e., strong consistency) for operations that can potentially conflict.

More recent research [13, 61, 14, 18] attempts to bring invariants to weak consistency scenarios. Escrow techniques [13, 65] enable sync-free execution until a certain limit at each replica, albeit synchronization is required to reset the limit. This is suitable for numeric invariants and is supported by PotionDB through the Bounded Counter [13]. Antidote SQL and IPA [61, 14] propose two different approaches for upholding uniqueness, referential integrity and disjunction invariants. Very recently, Borrego et. al. [18] propose the No-Op framework, leveraging on CRDTs to disable the effects of conflicting operations. It handles common kinds of invariants and is very expressive, but lacks scalability. Numerical and aggregation invariants are still challenging - IPA and No-Op struggle with them, being extremely restrictive on the concurrency allowed. Antidote SQL avoids this issue by using escrow techniques, however coordination is required in certain situations.

2.4.4 Existing systems

Suggestions for a better title? Also does this need some sort of introduction?

Spanner. Spanner [29] is a distributed, strongly consistent database proposed by Google for wide-area replication. The authors claim that data can be geo replicated while controversially claiming most applications only deploy data centers in a single continent. Data is multi-version and can be partially replicated, thus reducing replication costs.

Spanner's timestamps are based on real clock time, enabling lock-free reads in the past. Spanner uses Google's TrueTime API, which provides real time clocks with bounded uncertainty by levering on GPS and atomic clocks. Reads across partitions may still need to halt (or do extra communication) due to having to wait for the uncertainty.

Spanner deploys Paxos for strongly consistent replication and shards data internally across Paxos groups. Cross shard transactions require 2-phase commit (2PC), limiting fault tolerance and, due to multiple round-trips (RTTs), results in high latency, forcing data centers to be kept close (e.g. same continent). Furthermore, network partitions may halt the system. It is thus not suitable for geo-distributed applications requiring low latency.

MDCC. MDCC [46] is a geo-distributed strongly consistent system that aims to provide geo-distributed commits with a single RTT, unlike in the commonly deployed 2PC. Single RTT commits is desirable as RTTs between faraway data centers have high latency. For this goal, MDCC relies on concepts such as operation commutativity, having a master replica per object and combining multiple Paxos variants, namely Generalized, Multi and Fast Paxos. When the master is in a faraway data center, MDCC deploys "fast quorums" to still achieve single RTT commit without involving the master, thus reducing latency.

MDCC still has its shortcomings. It always requires at least one RTT, while some weak consistency systems execute operations locally. Furthermore, if two transactions do non-commutative writes to the same item (e.g., writes to a register), three RTTs are required for commit, which is unacceptable for many applications. Thus, MDCC is not appropriate for scenarios with popular items (e.g., social media, e-commerce).

SLOG. SLOG [69] is a geo-distributed, strongly consistent system that aims to provide low latency and high throughput. SLOG proposes using data locality to reduce RTT duration. Each data item has a master replica; writes and linearisable rads must be directed to it (snapshot reads can be at any replica). If all data accessed in a transaction is mastered in one data center, SLOG commits without cross data center coordination, achieving low latency commit (assuming the client is also nearby). However, high latency is incurred if clients are faraway from the master - even if a nearby replica also holds the data - or if a transaction spans masters in different data centers.

SLOG offers high throughput even under high contention, as well as automatic data remastering without considerable throughput impact. These properties come at a cost - clients cannot abort a transaction after submitting; transactions must be deterministic and must be fully known at the planning phase - i.e., dynamic transactions are not supported.

Caerus. Caerus is a strongly consistent, geo-distributed deterministic database able to commit geo-distributed transactions with a single RTT, by leveraging on transaction's determinism and read-write sets.Furthermore, Caerus relies on data locality and primary regions to reduce latency of single region transactions. Single RTT commits in all scenarios is ingenious, albeit it comes with important limitations. First, a single RTT can still be too slow in geo-distributed scenarios. Furthermore, fault tolerance is severely limited - to achieve low latency for single region, the authors limited fault tolerance to a single region, which is unrealistic due to network partitions. Finally, pre-emptively knowing read-write sets of transactions is not always feasible.

Detock. Similarly to Caerus, Detock proposes to use data locality and transaction determinism to provide strongly consistent, low latency single-region transactions and single RTT multi-region commits. Detock's transactional protocol also leverages on dependency graphs and deterministic deadlock resolution. However, to keep latency low, replication is only synchronous within a region, so a whole region failure (or network partition) may render data unavailable. Their evaluation only considers single and two region transactions, which is unrealistic when worldwide or statistical queries are considered. Additionally, throughput (but not safeness) is influenced by clock skew. Finally, they do not support partial replication, leading to high replication costs.

Dynamo. Dynamo [31] is a weakly consistent, highly available key-value store designed by Amazon. It provides eventual consistency with a simple get/put interface. Dynamo

leverages on several features to achieve low latency and large scale resilience. Data is partitioned with a zero-hop DHT - every node knows how to reach all data items. Dynamo embraces hardware heterogeneity - powerful servers act as multiple nodes. Each object is replicated in N nodes, while reads/writes involve R/W replicas - these parameters are configurable for different performance and safety trade-offs. Dynamo detects write conflicts and keeps all conflicting versions, returning them on read for the application to merge. According to Amazon's practical experience, state divergence rarely happens.

Dynamo applies a fully decentralized P2P structure, with gossip used for membership changes, failure detection and building the DHT. Dynamo focuses on 99.9% latency, while ensuring resilience, high availability and data durability. However, eventual consistency, no automatic data merge or object abstractions makes it hard to develop for Dynamo.

COPS. COPS [59] is a weakly consistent key-value store for the wide-area. COPS defines and provides causal+ consistency. Data is sharded within each data center (but not across), and causal relationships are supported between objects in different shards. An extension, COPS-GT, provides read transactions for reading multiple objects in the same version.

Operations in a datacenter are linearisable, and causally consistent across datacenters. As COPS require significant metadata (COPS-GT requires even more), the authors employ multiple garbage collection techniques. Providing read transactions comes at a cost - COPS-GT has reduced throughput compared to COPS in some scenarios, and COPS-GTs metadata may grow unbound under network partitions. The authors address the lack of write transactions, read+write transactions and richer datatypes in Eiger [60]. However, neither COPS nor Eiger offer partial replication.

Should I describe Cure in higher detail since PotionDB is based on Cure?

Cure. Cure [5] is a geo-replicated key-value store providing causal consistency. Cure focus on being always available while still providing as many guarantees and ease of use as possible. Cure provides highly available, causally consistent read-write transactions, with multiple data types implemented as CRDTs, offering a rich interface and automatic conflict solving. Cure employs optimizations to ensure non-blocking reads and that consecutive snapshots represent newer states even under failures. However, in some scenarios, performance is reduced compared to other solutions, and clients may block when switching to a different data center due to a failure. It also does not support partial replication nor other useful utilities such as views.

Walter. Walter [77] is a geo-replicated key-value store with partial replication and PSI (see Section 2.2.2) consistency. PSI itself is also defined in [77]. Walter deploys two interesting features: preferred sites, where writes of an object are more efficient; and counting sets (csets), a set where all operations are commutative (similar to set CRDTs). Transactions in a given site can be fast committed without synchronization if all updated objects are either csets or preferred on said site - thus, applications with good data locality and relying on csets may have good performance with Walter. However, transactions

updating objects with different preferred sites fall back to 2PC, requiring multiple RTTs and being partition-prone - i.e., the same limitations as in strongly consistent systems.

Should I add here a comment pointing out how Walter basically shows how useful commutativity is for avoiding synchronization?

Noria. Noria [35] is a streaming data-flow system for web applications that tolerate eventual consistency. Noria addresses limitations of both data-flow systems and databases, namely, windowed data for the former and materialized views for the latter. The main contribution is their partially-stateful data-flow model, allowing to only maintain partial operator state yet serve queries on all data. New entries are automatically added to the partial state while unused entries are evicted. Noria addresses how to efficiently provide queries and updates on such a system model.

Noria provides incrementally maintained, partially materialized views. The authors claim it is feasible to provide materialized views for every query with Noria. Noria delivers impressive performance, but with some shortcomings. First, state eviction from partial state is random, which is sub-optimal. Parametrized range queries (e.g., day > 10) and multi-column joins are unsupported. Replication and geo-distribution are not addressed - both essential for many web services nowadays. The authors claim application development is easier compared to cache + database setup, due to not needing manual caching - however eventual consistency complicates application development. Finally, some cross-shard operations are inefficient.

Chronocache. ChronoCache [36] is a mid-tier caching system for geo-distributed scenarios. It stores query results automatically and attempts to predict upcoming queries, executing them before they are issued by clients. Clients connect directly to ChronoCache instances deployed on nearby edge nodes, reducing client latency. ChronoCache does not use offline training, instead analysing clients' query patterns, building dependency graphs and analysing correlations. ChronoCache detects queries executed consecutively, as well as in loops, predictively executing multiple queries when one is issued by a client.

ChronoCache speeds up applications with constant access patterns. However, ChronoCache is not suited for applications with considerable update ratios or less repetitive access patterns. It is still dependent on the underlying database, so network partitions may halt the system.

TxCache. Traditional caching solutions provide only eventual consistency and no transaction semantics. TxCache, by contrast, provides strong consistency, supporting transactions with snapshot isolation and causality guarantees. TxCache provides a consistent, albeit likely stale, view of the database. Users can set a staleness limit, and it is easy to use as applications only need to mark functions as "cacheable". TxCache internally keeps a validity range for each cached result to indicate which snapshots can results be used on.

TxCache has considerable limitations. First, it requires significant database modifications to support reads on specific versions and notify TxCache of data invalidity. Read-write transactions execute directly on the database to avoid anomalies, thus taking

no benefit from caching. Also, only deterministic functions with no side effects and independent of external factors can be cached, missing possible records that could be safely cached. Required database modifications are non-trivial. Index or sequential scan queries lead to cacheable data that will be invalidated with any update to relevant tables, despite possible conditions on the queries allowing for smarter eviction strategies.

DBToaster. DBToaster [45] is a database focused on managing large datasets with a constant update stream and long-running analytical queries. DBToaster provides materialized views with incremental view maintenance, which can be automatically generated and potentially split in parts for efficiency. By relying on delta-queries and view splitting, the authors define triggers which update most views in constant time per update, avoiding costly loops and table joins. Query re-writing rules are also provided alongside an heuristic optimizer to further optimize views and their maintenance.

DBToaster shows that complex queries can be efficiently answered with materialized views and have affordable maintenance costs. However, some queries with nested aggregates still need to do a full view rebuild, and aggregates other than sum are not efficiently supported. Queries with inequality joins still need expensive joins to update the view.

DBSP. In DBSP [20], the authors define a model for incremental view maintenance by streaming database snapshots. A snapshot is defined as a cumulation of transactions. DBSP is expressive enough to support streaming window queries, queries on nested relations and recursive queries. Some SQL operators are efficiently supported, however common queries patterns (as seen TPC-H benchmark [30]) such as joins, distinct and aggregations with group by are costly in DBSP. Furthermore, DBSP requires transaction linearisability and all data to be available, which is challenging and limiting for geo-replicated scenarios.

2.4.5 Data access in PotionDB

PotionDB adopts a key-value interface, but vastly extends it by supporting multiple rich CRDTs. PotionDB exposes to the clients multiple read and update operations appropriate for each CRDT type, including read operations of partial state (e.g., lookup(e) in a set). Furthermore, PotionDB provides highly available, causally consistent transactions.

Should I mention here again PotionDB's consistency guarantees (PSI+cset)? I already mentioned this before at the end of Section 2.2. The motivation to mention it here again is to put it in comparison with the systems just mentioned before.

PotionDB focuses on supporting recurrent queries with very low latency, akin to reads on basic objects. Analytical and OLAP queries fit PotionDB's use case, as they are often recurring (e.g., checking the most sold products), benefit from low latency (e.g. to enable timely business decisions) and should ideally have negligible impact on the throughput and latency of other queries and updates. We propose to support this with materialized views that are automatically and incrementally maintained, even if the data referred by a view is partitioned across different data centers. We detail our solution in the next chapter.

3

Research Statement

Our work focuses on providing a scalable solution for data replication and management for a geo-distributed scenario. As discussed in Chapter 1, existing systems face scalability, performance and cost issues due to ever growing user bases and data sizes. In Chapter 2, we highlighted several techniques and algorithms that can be used to address these limitations, e.g. partial replication, non-uniform replication, causal consistency, etc. Moreover, analytical queries are essential nowadays to enable timely business decisions, requiring real-time analysis of large data volumes without visible system impact. However, as noted in Section 2.4, current solutions suffer from severe drawbacks, such as (i) using stale, incomplete or even inconsistent data; (ii) slow query execution or with impact on system performance; (iii) expensive infrastructures; (iv) costly synchronization, etc.

We plan to address the aforementioned challenges alongside others discussed in the following sections. We propose a multi-layered solution that will interleave both existing algorithms and techniques, as well as novel ones that we will design and propose. Our vision is to empower global-scale web applications that deploy multiple geo-distributed data centers with a scalable solution. We aim to minimize replication costs through data locality, partial replication and non-uniform replication, which together with casual consistency will enable cross data center scalability. Within each data center, scalability will is ensured through sharding. Additionally, our solution will answer recurring analytical queries with very low latency and without disrupting ongoing operations, by leveraging on materialized views. Our materialized views will be automatically and incrementally maintained - even over partitioned data - for ease of use and efficiency.

We highlight the novelty of supporting materialized views in a geo-distributed, weakly consistent and partially replicated setting. This is particularly challenging not only due to weak consistency, but also as our views must be able to relate to data that is partially replicated, even if no single server has all required data to compute the view. Finally, combining partial replication with non-uniform replication and causal consistency is also novel, as is the integration of views with those. Notably, our views will enable complex, recurring analytical queries concerning large sums of partially replicated data to run with latency comparable to basic object reads.

All contributions resulting from our work will be implemented in PotionDB, a geo-distributed replicated database being developed in the context of this work. PotionDB will be used both to showcase our concepts and to evaluate their performance. At the end, PotionDB should demonstrate our vision and the viability of our proposed solutions.

In the next sections we detail each of our contributions and the main pillars of work needed to achieve our goals. First, Section 3.1 describes the requirements and our solution for efficient support of recurring analytical queries through materialized views. Afterwards, Sections 3.2 and 3.3 describe, respectively, our replication and transactional algorithms. Section 3.4 describes our new solution for version management, essential to ensure data consistency. Section 3.5 describes our plans to support different consistency guarantees for views and objects, as well as data safety. Finally, 3.6 summarizes our contributions, both the ones already achieved and expected ones.

3.1 Recurring Queries and Views

In this work we aim to provide algorithms and a system in which recurring, analytical queries execute in near real-time alongside other "normal" queries. Analytical queries may require the analysis of millions of data objects to compute the answer, thus a dedicated solution is required.

We propose using materialized views to answer analytical queries. A materialized view contains a precomputed answer to a query, thus avoiding extensive data analysis or complex calculations. Bringing materialized views to a weakly consistent, geo-distributed setting is novel to the best of our knowledge, and introduces the following main challenges:

- Views must be kept up-to-date efficiently - an update to one object should not lead to a total rebuild of a view;
- Storage and update cost of views must be minimized - as discussed in Section 2.4, both factors have hindered the adoption of materialized views in commercial databases and remain an hot topic of research;
- Access to views must be efficient - querying a view should have the same order of complexity as querying other objects. Thus, all required data should be readily available without any complex or time-consuming computations or communications;
- Changes to objects and their related views must be atomic - it should not be possible, in the same transaction, for a client to observe a state in which the object was already updated but one or more of its views were not (and vice-versa). Views must stay consistent at all times with the objects they refer to;
- In each server where a given view is replicated, the view must reflect all updates to it even for entries whose objects are not locally replicated, without breaking causality.

We tackle the aforementioned problems as follows. To keep views up-to-date efficiently, we rely on incremental view maintenance - more precisely, an update to an object generates

one update for each relevant view, changing just the necessary data in the view without having to recalculate it from the ground up. In our PotionDB prototype, views are implemented as CRDTs like other objects - thus updating a view corresponds to a CRDT update. Since views are CRDTs, view queries are quick - the necessary data to answer the query is already present and should be supported by employing an appropriate CRDT. For example, for a top-k query, a top-k CRDT is used. For a query that requires multiple sums, with one sum per entry, a map CRDT with embedded counter CRDTs can be used.

Storage and update costs are a big concern with views. Views are expected to be replicated in multiple (potentially all) data centers and may concern large sums of data. Thus, it is essential to minimize both storage used and number of updates executed for each view in each data center. We tackle this two-fold. First, by using appropriate CRDTs, we can avoid storing unnecessary data. E.g., in a query for top 10 most sold products, we may only need the product's name and sales number for the query, so it is unnecessary to duplicate other product data. Secondly, by combining non-uniform and partial replication (Section 2.3), we can reduce storage and update costs further. Non-uniform replication identifies which updates may affect the state observable by the query, replicating only such updates. E.g., in a top 100 view, changes to objects outside the top may not affect the query result. This effectively reduces the number of updates applied and replicated by each replica, as well as the number of entries in the view. Partial replication further reduces costs, as some views may be replicated only in a subset of servers (e.g., a view concerning only Portuguese data does not need to be replicated everywhere).

The last two challenges are deeply related with the replication and transaction execution algorithms described in the next sections. Thus, we focus now on their implications. Ensuring views and their objects stay consistent has implications mostly on the transactional algorithm. First, it implies that transactions must be atomic - the client either observes all changes, or none. Furthermore, it must be ensured that when an object is updated, all relevant views are updated in the same transaction. Finally, to reduce the amount of concurrency anomalies that may be observed due to weak consistency, transactions must execute and be applied according to causal consistency. This ensures that, e.g., if two transactions execute in a row and modify the same view, the client observes both changes properly.

The last problem is mostly related with the replication algorithm. When a transaction is executed and is prepared to be replicated, it must be ensured that any view updates are sent to all replicas replicating said view(s), even if the base objects are not. This will imply some form of identifying the view(s) updates, sending them to the relevant replicas alongside all other updates relevant to each of those replicas.

3.2 Replication algorithms

Another crucial aspect of our work is the definition of replication and transaction execution algorithms that are appropriate for geo-replication. Both algorithms must cope and, when

possible, leverage on partial and non-uniform replication for efficiency. They also must guarantee views' well functioning as explained in the previous section.

For our replication algorithm, we define the following goals:

- Appropriate for long distance replication - this implies avoiding blocking mechanisms or having to coordinate with faraway replicas to make progress;
- Data efficient - in general, reduce the amount of data sent to each server.
- Consistent - must ensure consistency between objects is maintained. Namely, views must stay consistent with their related objects. Furthermore, views must be kept up-to-date, even if objects relevant for the view are not replicated in some of the servers where the view is.

Our replication algorithm is asynchronous. Thus, servers can continue executing operations while replication happens in the background, without affecting correctness, even if latency between servers is high. Transactions commit locally, without needing any synchronization with other servers. Thus, the replication process is all non-blocking.

We incorporate partial replication in our algorithm by only replicating, to each server, operations for objects the target server replicates. We do so as follows. First, we assume objects (including views) are arranged into "groups", which we call *buckets*. Each object can be assigned to one bucket, and each server can replicate one or more buckets. Each server must register which buckets they are interested in (e.g., by using a publish-subscribe service such as RabbitMQ [68]). When replicating a transaction, the transaction is split into parts - one for each bucket present in the transaction. Each part is sent to all replicas of the part's bucket. After all parts are sent, a message is sent to all involved servers signalling that the transaction is complete. Note that the mechanism just described ensures views are kept up-to-date even for updates originated by objects not replicated in the target replica. This is ensured as, for each transaction, all operations for the bucket of the view will be grouped and sent to all replicas of the view's bucket.

There is actually an optimization possible, related to sending the special message signalling the end of the transaction - when we send many transactions in a group, we only need to send this special message on the last one. Should this be mentioned?

To further reduce storage and network costs, specially for views, our replication algorithm must also support non-uniform replication, which has implications in the algorithm's design. In non-uniform replication, some operations that were previously not relevant (i.e., did not need to be replicated), may need to be replicated due to the execution of another operation. E.g., a remove of an element in the top 100 may force an update to some element not in top 100 to be replicated, as a new element may be joining the top. Supporting non-uniform replication in our design has two implications. In terms of the replication algorithm, whenever a transaction with non-uniform objects is executed, any "new" operations generated by executing the transaction are propagated to relevant replicas grouped in one transaction. It also has implications on the transactional algorithm, which we detail in the next section.

Finally, our replication algorithm must maintain consistency. We aim to provide causal consistency across servers. Thus, our replication algorithm ensures operations are sent

and received according to causal order. In an implementation, this can be ensured by using, e.g., vector clocks. Applying the operations correctly according to causal order, as well as dealing with the visibility of the operations, is responsibility of the transactional algorithm described next.

3.3 Transactional algorithm

The transactional algorithm is the one responsible for applying both updates and queries, as well as keeping the system consistent. In more detail, our algorithm must ensure the following:

- All updates of a transaction must be made visible at the same time - for any query on another transaction, either all updates are visible or none are (atomicity);
- Queries must observe a consistent snapshot of the database - all queries of a transaction must observe objects in the same version;
- Transactions received though replication must be executed correctly - despite only receiving the parts of the transaction for objects locally replicated, consistency must still be ensured;
- Views and all related objects must stay consistent;
- Whenever possible, to make better usage of multi-core CPUs, transactions should execute concurrently inside a replica as long as no conflict arises from doing so.

We assume internally the system is partitioned in an arbitrary number of "partitions", with each partition storing a subset of the keyspace of objects. This allows to explore parallelism, by either assigning different partitions to different servers or to different CPU cores. In our work, we assume the later. Our transactional algorithm ensures, at the level of a single server, snapshot isolation (SI). When considering consistency between servers, our protocol ensures causal consistency.

Do I need to mention that partitions != buckets? Or not necessary? If so, how should I mention that?

The existence of partitions lets read-only transactions execute in parallel. In fact, it allows even for queries inside the same transaction to execute in parallel. Both factors speed up the execution of transactions considerably. This is possible as, associated to each transaction, is a version identifier (e.g., a vector clock). When executing queries, each partition uses this version identifier to ensure the correct version of each object is read (more details on version management in the next section). Thus, queries always observe a consistent snapshot of the database.

On the other hand, read-write transactions may have to coordinate with other transactions to prevent conflicts. Our algorithm uses a two-phase commit protocol (2PC, check Section 2.4.4) to coordinate transactions when updates are involved. Given the partitions are in the same machine, the protocol executes very quickly, as the communication is only between threads. Our algorithm ensures the coordination is only done between partitions

involved in read-write transactions - other partitions may keep executing read-only transactions unaffected. In fact, if two read-write transactions operate in distinct subsets of partitions, they will execute in parallel.

The usage of 2PC and versions for objects ensures all updates of a transaction are visible at a single point in time. When a transaction is committed, the updated objects are moved to a new version, correspondent to the transaction's commit clock. This new version is only made available to be used by other transactions to read after all partitions finish executing the commit. Thus, the new version becomes visible at the same time in all partitions. The consistency between views and their objects is provided by requiring updates to objects and their view(s) to be grouped in the same transaction, ensuring both the view's and the objects' updates become visible at the same time.

In transactions received through replication, only some parts of the transaction are received - the parts correspondent to the buckets locally replicated. This poses a challenge as the transaction must be correctly rebuild in order to ensure the end state is the same as if the full transaction had been received. Note that as long as updates to the same object are executed by the same order, the final state of each object will be the same. Also note that if all updates are made visible at the same time, then consistency is also ensured. This is due to the fact that updates to different objects are independent - executing the updates to object A before object B, or the other way around, leads to the same results. Thus, to correctly rebuild the transaction, all that is needed is for updates for the same object to keep the original order. The replication algorithm ensures this, as all updates to the same object belong to the same bucket, and the order of updates in a bucket is kept during replication. As such, a simple transaction rebuild algorithm can consist in grouping all buckets received, with all updates in a bucket according to the original order, and the order between buckets can be any. After a received transaction is rebuilt, it is executed as any other transaction, as long as all the transactions that happened-before have already been executed (if not, the transaction is put on hold until it can be executed).

Is the paragraph above clear?

To support the execution of transactions with objects supporting non-uniform replication, we proceed as follows. When a transaction with non-uniform objects is executed and generates new operations, those operations are grouped in a new transaction, from the partitions involved. After the new transaction commits in the relevant partitions, the transaction is sent to the replication algorithm.

A current limitation of our work, related with both the replication and transactional algorithm, is that buckets are static. That is, it is assumed the set of buckets a server replicates does not change. However, as seen in Section 2.3.3, for some applications access patterns may change as time goes by, or even data locality itself (e.g., an user which travels to another region). We plan to address this by supporting what is known as dynamic partitioning. This requires designing a protocol that allows for the following:

- support for new nodes joining the system;
- support for nodes leaving the system, as well as their recovery in case of a crash;

- efficient data transfers (transfer only relevant objects, compact operation history, etc.);
- dynamic partitioning itself - add or remove buckets from existing servers.

The last goal is particularly challenging, as the repartitioning may happen while multiple transactions are ongoing, potentially for the buckets being added/removed. We must ensure no update is lost during repartitioning and that queries can still be served. Consistency between views and objects must also be maintained even if views or objects referred by views are being re-partitioned. At the moment, support for new nodes joining the system and a basic data transfer is already supported, but more work on the protocol is needed to achieve the remaining goals.

3.4 Version Management

In order to ensure each transaction can see a consistent snapshot of the database, while still allowing multiple transactions to occur in parallel, it is necessary to provide specific versions of any given object. More precisely, it is necessary to provide the version required by any ongoing transaction. Thus, some mechanism to provide old versions is needed.

Some solutions, e.g., Cure, keeps old versions around, garbage collecting them when no longer needed [5]. In this work, we decided to explore a different approach - instead, we keep only the latest version (and optionally, we may cache a few other versions for performance when appropriate, but not as a requirement). Alongside the latest version, we keep two logs for each object - the list of updates executed, and the list of "effects" of each update. We define an effect as the changes an operation had on an object when it was executed. Both logs are needed, as the same operation may imply different changes to the object depending on when it was executed. E.g., a remove from a set may or not remove an object from the set, depending on the existence of concurrent adds for the same element.

Our algorithm thus works as follows. First, given a target version and the current version, we "undo" the effects, until we are at a version for which the vector clock is smaller or equal in every entry than the target version. Afterwards, we re-apply the operations needed to reach the target vector clock. The reason for the need to re-apply operations is due to concurrent operations. Operations that are concurrent can be re-applied in any order, but causality must still be maintained between two non-concurrent operations (e.g., operation A may be concurrent to operation B and C, but B and C are causally related - A can be re-applied at any time, but B and C must be applied in order).

As of now, PotionDB only implements the version management described above. On the next steps of our work, we plan to implement another version management system which is based on keeping in memory the previous versions of the objects. The intention is to then compare the performance of both solutions and access the advantages of ours, namely in terms of performance and space used. Furthermore, and before the

Is this explanation of the version thing okay? Also, should I include some scheme/image with an example?

comparison is done, it is also necessary to design a garbage collection algorithm for version management. Without garbage collection, the metadata of both solutions - respectively, the log of updates + effects and the list of old states, will keep growing indefinitely. Adding support for efficient garbage collection is not straightforward - it is necessary to determine how old the information must be in order to be safe to be garbage collected. If information too recent is deleted, ongoing transactions may have to be aborted (as they need garbage collected versions) and thus affect performance and user experience negatively.

One way to address this is to, periodically, delete old information (old updates and effects) that are guaranteed to no longer be necessary. This can be achieved by checking which partition has the oldest clock (or by keeping track of a clock that is known to be "safe" in all partitions), and deleting logs of updates and effects older than said clock. It will be necessary to consider ongoing transactions, possibly by delaying the deletion of old data, using an even older clock or restarting the transaction with a more recent clock. Finally, the cleaning should be done when system's load is low, or in the background to avoid affecting performance.

3.5 Consistency levels

The text above may need to be updated. We already support the other version, we have GC but both have not been evaluated for performance.

Currently, our algorithms are designed to provide causal consistency. Our prototype implementation supports both causal consistency and read committed consistency, albeit only one can be used at a time for all transactions. Read committed provides less guarantees but allows for higher operation throughput. However, we believe more work with consistency could be interesting research, due to the implications views have in our system.

One possibility would be to support multiple consistency levels for transactions. This would imply supporting transactions executing with different consistency levels, as done e.g. with red-blue consistency. This would be useful, as some queries/transactions may be able to cope with weaker consistency guarantees and thus benefit from increased performance. Other queries that require stronger consistency can request so, executing a bit slower but with stronger guarantees on the result. One way to ensure this is to make the reads responsible for ensuring the desired consistency level. I.e., transactions with writes would still be atomic to ensure objects (and their views) stay consistent, while reads executing under lower consistency guarantees can, e.g., read objects at different versions without internal coordination to improve performance.

Another interesting possibility to strengthen our consistency guarantees would be by supporting invariants. Invariants allow users to define conditions for an object which must stay true at any point in time. This is useful to, e.g., ensure an item's stock never goes negative. Supporting this under weak consistency is challenging as servers execute operations concurrently without synchronization. Some solutions already exist in the literature, e.g., [13], which suggests for the case of numeric invariants in counters, to distribute among replicas how much they can decrement/increment safely without

breaking the invariant. A similar principle could potentially be used for other kinds of objects. Supporting invariants in materialized views would be extra challenging, as these invariants would have implications on the state of many other objects, which would need invariants too.

Work on strengthening/expanding our consistency guarantees has not been started yet. While this seems an interesting venue, it is still to be defined how will we precisely achieve said goal. The possibilities above are, indeed, some possibilities we may take, but it is not yet set on stone and is thus subject to changes.

I don't think I mentioned very well the part of "this area/topic seems interesting but it is not yet well defined". Suggestions? Maybe I should had taken out detail on the invariants/consistency levels? I thought they could be here as possible routes we may take, but maybe that is not wise to do at this point in time.

3.6 Contributions

In this section we now summarize both the contributions already achieved with this work, as well as the expected upcoming ones. As of now, we have done the following contributions:

- A new replication algorithm, combining partial, non-uniform and geo replication;
- A transactional algorithm with support for snapshot isolation at the server level, while coping with causal consistency across servers and with support for both partial and non-uniform replication;
- Support for materialized views under causal consistency, where updates to objects and their views are visible atomically;
- Support for materialized views to refer to data not locally replicated + non-uniform and partial replication of views to reduce their storage and replication costs;
- A new form of incrementally maintaining views, with CRDT update operations;
- An algorithm to reconstruct the state of any object at any given point in time, by using the actual object state, a list of operations and a list of effects;
- An implementation in our prototype PotionDB of all the features above.

We highlight the novelty of our replication algorithm, which supports both partial and non-uniform replication efficiently. Materialized views in a geo-distributed, weakly consistent, with partial and non-uniform replication support, without requiring all relevant objects to be present in the same server, is also novel, as is the incremental updating of views with CRDTs' update operations.

Albeit not a contribution on itself, considerable amounts of work have been put into supporting many kinds of CRDTs in PotionDB, to ensure many different kinds of views and objects can be provided. Not only this eases application development with PotionDB, it is essential to ensure as many kinds of queries as possible are supported. This also implied some extensions or adaptations to existing CRDT designs. Another noteworthy mention is the implementation of the TPC-H benchmark [30], namely its dataset, updates and a subset of its queries. As TPC-H is oriented to relational databases using SQL, it

was necessary to adapt the tables to objects of PotionDB, as well as define the appropriate views for each query. While we only implemented a subset of queries, we note that all queries in TPC-H would be possible to implement with materialized views in PotionDB using the existing CRDTs. The main goals of implementing TPC-H's benchmark is to both evaluate PotionDB's performance and show its expressiveness. As a last mention, we have also implemented a simple algorithm for executing transactions in a replica when some of the objects are not locally replicated.

Please tell me if the paragraph above should be deleted - I just wanted to show where some of the time was spent.

In terms of the work left to do, we expect to achieve the following contributions:

- A garbage collection algorithm for systems implementing SI with vector clocks. This algorithm will be able to determine a safe clock in all partitions and do the cleaning without affecting considerably the system's performance;
- Provision of different consistency guarantees, possibly by providing different consistency levels for transactions or invariants in a weakly consistent scenario
- An algorithm for dynamic partitioning compatible with snapshot isolation (server-level) and causal consistency (cross-servers), without loss of updates or stopping serving queries. This re-partitioning will work both for creating/removing partitions, adding/removing servers, and properly support views and their related objects;
- A performance evaluation of our system and of the algorithms implemented.

It is noteworthy that the aforementioned contributions always have to take in consideration the consistency between views and their objects, which makes these expected contributions more challenging and novel.

Any other contributions? Any that should be removed?

4

Work Plan

In this section we describe how we plan to make usage of the remaining time until the end of the PhD. Table 4 shows the distribution of time for each task left. We have described the challenges of each task in Chapter 3.

Time frame	Activity
June 2022 to September 2022	garbage collection for version management
September 2022 to November 2022	Implementation of alternative version management solution
September 2022 to January 2023	Evaluation of garbage collection and version management solutions
November 2022 to May 2023	Dynamic partitioning
March 2023 to June 2023	Evaluation of PotionDB with dynamic partitioning
February 2023 to November 2023	Consistency levels
July 2023 to December 2023	Evaluation of consistency levels + final PotionDB evaluation
September 2023 to March 2024	Writing of PhD thesis and final publications

A lot of work has already been done in the context of evaluating PotionDB, namely by implementing TPC-H's benchmark and other test clients. We believe this will make the next evaluating steps quicker and smoother, as the testing setup can be leveraged on for the next steps. Our evaluation will be done in the Grid'5000 testbed, which allows to make usage of multiple machines to run instances of PotionDB on. Latency from geo-distribution can be simulated by imposing latency on the links, which was already done in our experiments.

There is some considerable overlap between dynamic partitioning and consistency levels. This happens as supporting different consistency models and/or invariants may have impacts on how dynamic partitioning works, thus we need to make sure our partitioning algorithm can cope with different consistency guarantees.

Results obtained during the course of this work will be published in top-venue conferences. At the moment, a submission to VLDB is being prepared, hence the extensive evaluation work already done. We plan to do other submissions to other conferences.

Do I need to mention here the challenges/difficulties to address? I feel like that was already mentioned in the Research Statement.

I will make some scheme similar to the timeline for the grant when this is more well defined. The current table is just temporary.

Should I in the plan mention about our submission to VLDB? Like have that be part of the table/figure?

A concern of mine is that adding "consistency levels" may imply changes to dynamic partitioning.

Do I need to make any special reference

Interesting publication targets besides VLDB include, for example, OSDI, SOSP, EuroSys, PODC and DISC.

Bibliography

[1] M. Abebe, B. Glasbergen, and K. Daudjee. "MorphoSys: automatic physical design metamorphosis for distributed database systems". In: *Proceedings of the VLDB Endowment* 13.13 (2020), pp. 3573–3587 (cit. on p. 13).

[2] V. Abramova and J. Bernardino. "NoSQL databases: MongoDB vs cassandra". In: *Proceedings of the international C* conference on computer science and software engineering*. 2013, pp. 14–22 (cit. on pp. 16, 17).

[3] M. Adorjan. *CloudPing*. https://www.cloudping.co/grid. Accessed on 27th May 2025. 2015 (cit. on p. 14).

[4] R. Ahmed et al. "Automated generation of materialized views in Oracle". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3046–3058 (cit. on p. 17).

[5] D. D. Akkoorath et al. "Cure: Strong semantics meets high availability and low latency". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 405–414 (cit. on pp. 4, 7, 8, 10–12, 14–16, 20, 29).

[6] P. S. Almeida, A. Shoker, and C. Baquero. "Efficient state-based crdts by delta-mutation". In: *International Conference on Networked Systems*. Springer. 2015, pp. 62–76 (cit. on p. 12).

[7] S. Almeida, J. Leitão, and L. Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, pp. 85–98 (cit. on pp. 7, 8, 14).

[8] R. Alotaibi et al. "HADAD: A lightweight approach for optimizing hybrid complex analytics queries". In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 23–35 (cit. on pp. 2, 16, 17).

[9] K. Amiri et al. "DBProxy: A dynamic data cache for Web applications". In: *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE Computer Society. 2003, pp. 821–821 (cit. on p. 17).

[10] J. E. Armendáriz-Inigo et al. "SIPRe: a partial database replication protocol with SI replicas". In: *Proceedings of the 2008 ACM symposium on Applied computing*. 2008, pp. 2181–2185 (cit. on pp. 2, 10–13).

[11] P. Bailis et al. "Coordination avoidance in database systems". In: *Proceedings of the VLDB Endowment* 8.3 (2014), pp. 185–196 (cit. on p. 18).

[12] J. Baker et al. "Megastore: Providing scalable, highly available storage for interactive services". In: (2011) (cit. on p. 14).

[13] V. Balegas et al. "Extending eventually consistent cloud databases for enforcing numeric invariants". In: *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2015, pp. 31–36 (cit. on pp. 18, 30).

[14] V. Balegas et al. "IPA: Invariant-preserving applications for weakly-consistent replicated databases". In: *arXiv preprint arXiv:1802.08474* (2018) (cit. on pp. 17, 18).

[15] V. Balegas et al. "Putting consistency back into eventual consistency". In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–16 (cit. on pp. 17, 18).

[16] C. Baquero et al. "Composition in state-based replicated data types". In: *Bulletin of the European Association for Theoretical Computer Science* 123 (2017) (cit. on p. 12).

[17] N. M. Belaramani et al. "PRACTI Replication." In: *NSDI*. Vol. 6. 2006, pp. 5–5 (cit. on pp. 12, 13).

[18] D. Borrego et al. "Ensuring Convergence and Invariants Without Coordination". In: *39th European Conference on Object-Oriented Programming (ECOOP 2025)* (cit. on p. 18).

[19] M. Bravo, L. Rodrigues, and P. Van Roy. "Saturn: A distributed metadata service for causal consistency". In: *Proceedings of the Twelfth European Conference on Computer Systems*. 2017, pp. 111–126 (cit. on pp. 4, 14).

[20] M. Budiu et al. "DBSP: Automatic Incremental View Maintenance for Rich Query Languages". In: *Proceedings of the VLDB Endowment* 16.7 (2023), pp. 1601–1614 (cit. on pp. 17, 22).

[21] S. Burckhardt, A. Gotsman, and H. Yang. *Understanding Eventual Consistency*. Tech. rep. MSR-TR-2013-39. 2013. URL: https://www.microsoft.com/en-us/research/publication/understanding-eventual-consistency/ (cit. on pp. 4, 6–9, 11, 12).

[22] D. Buytaert. *Faster is better*. https://dri.es/faster-is-better. Accessed on 27th May 2025. 2009 (cit. on pp. 13, 14).

[23] G. Cabrita and N. Preguiça. "Non-uniform Replication". In: *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS 2017)*. 2017 (cit. on p. 15).

[24] G. M. Cabrita. "Non-uniform replication for replicated objects". PhD thesis. 2017 (cit. on p. 15).

[25] S. J. Castiñeira and A. Bieniusa. "Collaborative offline web applications using conflict-free replicated data types". In: *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. 2015, pp. 1–4 (cit. on p. 10).

[26] D. D. Chamberlin and R. F. Boyce. "SEQUEL: A structured English query language". In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. 1974, pp. 249–264 (cit. on pp. 4, 16).

[27] Y. Chen et al. "TDSQL: Tencent Distributed Database System". In: *Proceedings of the VLDB Endowment* 17.12 (2024), pp. 3869–3882 (cit. on p. 7).

[28] B. F. Cooper et al. "PNUTS: Yahoo!'s hosted data serving platform". In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1277–1288 (cit. on pp. 8, 18).

[29] J. C. Corbett et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22 (cit. on pp. 2, 7, 11, 13, 14, 16, 18).

[30] T. P. P. Council. *TPC Benchmark H (Decision Support) Standard Specification Revision 2.18.0*. `tpc.org/tpch/`. 2018 (cit. on pp. 16, 22, 31).

[31] G. DeCandia et al. "Dynamo: Amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220 (cit. on pp. 1, 4, 7, 8, 11, 13–16, 19).

[32] L. Dong et al. "Marviq: Quality-Aware Geospatial Visualization of Range-Selection Queries Using Materialization". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 67–82 (cit. on p. 17).

[33] A. Fekete et al. "Making snapshot isolation serializable". In: *ACM Transactions on Database Systems (TODS)* 30.2 (2005), pp. 492–528 (cit. on pp. 7, 9).

[34] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *Acm Sigact News* 33.2 (2002), pp. 51–59 (cit. on p. 6).

[35] J. Gjengset et al. "Noria: dynamic, partially-stateful data-flow for high-performance web applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 213–231 (cit. on pp. 16, 17, 21).

[36] B. Glasbergen et al. "Chronocache: Predictive and adaptive mid-tier query result caching". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2391–2406 (cit. on pp. 13, 14, 17, 21).

[37] L. Glendenning et al. "Scalable consistency in Scatter". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 15–28 (cit. on p. 14).

[38] Gomez. *Why Web Performance Matters: Is Your Site Driving Customers Away?* Accessed on 5th September 2024. URL: https://montereypremier.com/wp-content/uploads/2019/10/201110_why_web_performance_matters.pdf (cit. on p. 1).

[39] J. Gonçalves, M. Matos, and R. Rodrigues. "SconeKV: A Scalable, Strongly Consistent Key-Value Store". In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 4164–4175 (cit. on pp. 7, 11, 13, 14).

[40] H. Gupta and I. S. Mumick. "Incremental maintenance of aggregate and outerjoin expressions". In: *Information Systems* 31.6 (2006), pp. 435–464 (cit. on p. 17).

[41] H. Gupta and I. S. Mumick. "Selection of views to materialize in a data warehouse". In: *IEEE Transactions on Knowledge and Data Engineering* 17.1 (2005), pp. 24–43 (cit. on pp. 2, 3, 16, 17).

[42] J. Hildred, M. Abebe, and K. Daudjee. "Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems". In: *Proceedings of the VLDB Endowment* 17.3 (2023), pp. 469–482 (cit. on pp. 1, 7, 11, 13, 14).

[43] G. Kaki et al. "Safe replication through bounded concurrency verification". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–27 (cit. on p. 18).

[44] B. Kate et al. "Easy freshness with Pequod cache joins". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 415–428 (cit. on pp. 15–17).

[45] C. Koch et al. "DBToaster: higher-order delta processing for dynamic, frequently fresh views". In: *The VLDB Journal* 23.2 (2014), pp. 253–278 (cit. on pp. 2, 16, 17, 22).

[46] T. Kraska et al. "MDCC: Multi-data center consistency". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, pp. 113–126 (cit. on pp. 1, 7, 11, 13, 14, 18).

[47] K. Krikellas et al. "Strongly consistent replication for a bargain". In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE. 2010, pp. 52–63 (cit. on pp. 7, 8, 10, 11).

[48] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40 (cit. on pp. 7, 8, 11).

[49] L. Lamport. "Paxos made simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58 (cit. on p. 11).

[50] L. Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 179–196 (cit. on p. 8).

[51] P.-A. Larson and J. Zhou. "Efficient maintenance of materialized outer-join views". In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 56–65 (cit. on pp. 16, 17).

[52] J. Lee et al. "Asymmetric-partition replication for highly scalable distributed transaction processing in practice". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3112–3124 (cit. on pp. 10, 12, 13).

[53] K. Y. Lee and M. H. Kim. "Optimizing the incremental maintenance of multiple join views". In: *Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP*. 2005, pp. 107–113 (cit. on pp. 16, 17).

[54] C. Li, N. Preguiça, and R. Rodrigues. "Fine-grained consistency for geo-replicated systems". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 359–372 (cit. on p. 18).

[55] L. Li et al. "BinDex: A Two-Layered Index for Fast and Robust Scans". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 909–923 (cit. on p. 17).

[56] P. Li et al. "LISA: A learned index structure for spatial data". In: *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2020, pp. 2119–2133 (cit. on p. 17).

[57] A. van der Linde, J. Leitão, and N. Preguiça. "δ-crdts: Making δ-crdts delta-based". In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*. 2016, pp. 1–4 (cit. on p. 12).

[58] A. van der Linde et al. "Legion: Enriching internet services with peer-to-peer interactions". In: *Proceedings of the 26th International Conference on World Wide Web*. 2017, pp. 283–292 (cit. on p. 10).

[59] W. Lloyd et al. "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 401–416 (cit. on pp. 4, 7, 8, 11, 13–16, 20).

[60] W. Lloyd et al. "Stronger Semantics for {Low-Latency}{Geo-Replicated} Storage". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 313–328 (cit. on pp. 1, 8, 10, 12–14, 16, 20).

[61] P. Lopes et al. "Antidote SQL: relaxed when possible, strict when necessary". In: *arXiv preprint arXiv:1902.03576* (2019) (cit. on p. 18).

[62] Y. Lu et al. "Epoch-based commit and replication in distributed OLTP databases". In: (2021) (cit. on pp. 11–14).

[63] D. Navalho, S. Duarte, and N. Preguiça. "A study of crdts that do computations". In: *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. 2015, pp. 1–4 (cit. on p. 15).

[64] C. D. Nguyen, J. K. Miller, and D. J. Abadi. "Detock: High Performance Multi-region Transactions at Scale". In: *Proceedings of the ACM on Management of Data* 1.2 (2023), pp. 1–27 (cit. on pp. 1, 7, 8, 11, 13, 14).

[65] P. E. O'Neil. "The escrow transactional method". In: *ACM Transactions on Database Systems (TODS)* 11.4 (1986), pp. 405–430 (cit. on p. 18).

[66] G. Oster et al. "Proving correctness of transformation functions in collaborative editing systems". PhD thesis. INRIA, 2005 (cit. on p. 10).

[67] D. R. Ports et al. "Transactional consistency and automatic management in an application data cache". In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010 (cit. on pp. 16, 17).

[68] RabbitMQ. *RabbitMQ*. https://www.rabbitmq.com/. May 2022 (cit. on p. 26).

[69] K. Ren, D. Li, and D. J. Abadi. "Slog: Serializable, low-latency, geo-replicated transactions". In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1747–1761 (cit. on pp. 7, 8, 11, 13, 14, 16, 19).

[70] A. Rijo, C. Ferreira, and N. Preguiça. "Set CRDT com Múltiplas Políticas de Resolução de Conflitos". In: *INForum 2018 - Actas do 10º Simpósio de Informática*. Universidade de Coimbra, 2018 (cit. on p. 10).

[71] H.-G. Roh et al. "Replicated abstract data types: Building blocks for collaborative applications". In: *Journal of Parallel and Distributed Computing* 71.3 (2011), pp. 354–368 (cit. on p. 10).

[72] R. Schulze et al. "ClickHouse-Lightning Fast Analytics for Everyone". In: *Proceedings of the VLDB Endowment* 17.12 (2024), pp. 3731–3744 (cit. on pp. 2, 3, 13, 16, 17).

[73] E. Schurman and J. Brutlag. *Performance Related Changes and their User Impact. Presented at Velocity Web Performance and Operations Conference*. http://slideplayer.com/slide/1402419/. 2009 (cit. on p. 1).

[74] M. Shapiro et al. "A comprehensive study of convergent and commutative replicated data types". PhD thesis. Inria–Centre Paris-Rocquencourt; INRIA, 2011 (cit. on pp. 10–12).

[75] P. Sioulas, I. Mytilinis, and A. Ailamaki. "Real-Time Analytics by Coordinating Reuse and Work Sharing". In: *arXiv preprint arXiv:2307.08018* (2023) (cit. on p. 2).

[76] P. Sioulas, I. Mytilinis, and A. Ailamaki. "Real-Time Analytics by Coordinating Reuse and Work Sharing". In: *arXiv preprint arXiv:2307.08018* (2023) (cit. on p. 17).

[77] Y. Sovran et al. "Transactional storage for geo-replicated systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 385–400 (cit. on pp. 4, 8, 9, 11, 14–16, 20).

[78] M. Stonebraker et al. "The end of an architectural era: It's time for a complete rewrite". In: *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 2018, pp. 463–489 (cit. on p. 13).

[79] C. Sun and C. Ellis. "Operational transformation in real-time group editors: issues, algorithms, and achievements". In: *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. 1998, pp. 59–68 (cit. on p. 10).

[80] D. B. Terry et al. "Session guarantees for weakly consistent replicated data". In: *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. IEEE. 1994, pp. 140–149 (cit. on pp. 4, 8).

[81] V.-D. Tran, H. Kato, and Z. Hu. "BIRDS: programming view update strategies in datalog". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2897–2900 (cit. on p. 17).

[82] N. VanBenschoten et al. "Enabling the next generation of multi-region applications with CockroachDB". In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 2312–2325 (cit. on p. 14).

[83] M. Wiesmann and A. Schiper. "Comparison of database replication techniques based on total order broadcast". In: *IEEE Transactions on Knowledge and Data Engineering* 17.4 (2005), pp. 551–566 (cit. on p. 11).

[84] M. Wiesmann et al. "Understanding replication in databases and distributed systems". In: *Proceedings 20th IEEE International Conference on Distributed Computing Systems*. IEEE. 2000, pp. 464–474 (cit. on p. 7).

[85] H. Xu et al. "BP-Tree: Overcoming the point-range operation tradeoff for in-memory B-trees". In: *Proceedings of the 2024 ACM Workshop on Highlights of Parallel Computing*. 2024, pp. 29–30 (cit. on p. 17).

[86] H. Yang et al. "'Cause I'm Strong Enough: Reasoning about consistency choices in distributed systems". In: *43rd ACM Symposium on Principles of Programming Languages (POPL 2016)*. Association for Computing Machinery. 2016 (cit. on p. 18).

[87] M. Zawirski et al. "Write fast, read in the past: Causal consistency for client-side applications". In: *Proceedings of the 16th Annual Middleware Conference*. 2015, pp. 75–87 (cit. on p. 10).

[88] C. Zhan et al. "Analyticdb: Real-time olap database system at alibaba cloud". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 2059–2070 (cit. on pp. 2, 3, 16).

[89] J. Zhou, P. Larson, and J. Goldstein. "Partially materialized views". In: *Technical Report*. 2005 (cit. on pp. 16, 17).

[90] J. Zhou, P.-A. Larson, and H. G. Elmongui. "Lazy maintenance of materialized views". In: *Proceedings of the 33rd international conference on Very large data bases*. 2007, pp. 231–242 (cit. on pp. 16, 17).

A

NOVAthesis covers showcase

This Appendix shows examples of covers for some of the supported Schools. When the Schools have very similar covers (e.g., all the schools from Universidade do Minho), just one cover is shown. If the covers for MSc dissertations and PhD thesis are considerable different (e.g., for FCT-NOVA and UMinho), then both are shown.

B

Appendix 2 Lorem Ipsum

This is a test with citing something [**ecoop12-dias**] in the appendix.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea

dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

I

Annex 1 Lorem Ipsum

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum

wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.