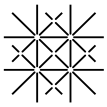


University of Basel

Freshness-aware Data Management in a Polystore system

Master's Thesis

Faculty of Science of the University of Basel
Department of Mathematics and Computer Science
Database and Information Systems Group
dbis.dmi.unibas.ch

Examiner: Prof. Dr. Heiko Schuldt
Supervisor: Marco Vogt, MSc.

Marc Hennemann
marc.hennemann@stud.unibas.ch
19-067-586

May 7, 2022

Acknowledgments

First of all, I would like to thank Prof. Dr. Heiko Schuldt for giving me the opportunity to conduct my Master's Thesis in the Database and Information Systems (DBIS) group.

I also want to thank my supervisor Marco Vogt, for the helpful discussions and his valuable feedback throughout this time. I could always rely on his advice and guidance, providing me with insightful input whenever I had a question.

Additionally, I would like to thank Daniel Haß and Lisa Gauthier for proofreading this thesis. Their review and comments helped to form and shape this thesis.

Finally, I want to express my deepest gratitude to my family and friends, for providing me with unfailing support. Their continuous encouragement allowed me to overcome every obstacle that I encountered over the years. This accomplishment would not have been possible without them.

Abstract

In recent years data has not only grown in volume and variety but also in importance. It has developed into an essential driver to advance industrial as well as scientific achievements. This influence elevated the need to store data reliably in different distributed locations, to ensure its availability even in the presence of partial failures.
But, since the cost of maintaining several replicas per data object is rather high, systems often need to compromise between access times and consistency. Therefore they usually limit the number of synchronously updated replicas, and lazily replicate the changes to the remaining nodes in a cluster. To cope with constantly varying requirements and the increased complexity of systems, efficient and cost-optimized data processing has become a crucial operational aspect to allow access to distributed data.
Consequently, to efficiently utilize all resources in a system, various notions of freshness have been established, to allow a more flexible approach to access distributed data. These provide the possibility to specify freshness demands and define whether the most recent data is required or if a request can be satisfied with slightly outdated data as well.

In this thesis, we summarize and define the fundamental features of freshness-awareness and exemplify the implementation on the basis of the Polystore system, Polypheny-DB.
We, therefore, establish a lazy replication algorithm to defer updates on selected replicas and propagate changes across the underlying stores, to create multiple versions of data objects. Accompanied with individual query extensions and various freshness metrics, we allow to centrally process these possibly outdated objects and leverage the key benefits of the encompassed stores, to increase the overall efficiency of the system.

We will evaluate the individual functional building blocks, necessary to establish freshness-awareness, to identify the impact the implementation has on the system. Further, we want to observe how the key functionalities of the underlying stores, influence the behavior of the solution.
The results show that although the solution is not universally applicable to every scenario, it significantly improves the overall execution, especially in distributed Polystore systems.

Table of Contents

Acknowledgments ii

Abstract iii

1 Introduction 1
- 1.1 Polystores . . . 2
- 1.2 Motivation . . . 2
- 1.3 Contribution . . . 4
- 1.4 Outline . . . 4

2 Related Work 5
- 2.1 Data Freshness . . . 5
- 2.2 Freshness Analysis and Metrics . . . 6
- 2.3 Freshness Constraints . . . 6
- 2.4 Update Propagation . . . 7
- 2.5 Refresh Strategies . . . 8
- 2.6 Consistency . . . 9
- 2.7 Freshness-aware Read Access . . . 10

3 Foundations of Distributed Data Management 11
- 3.1 Data Partitioning . . . 11
- 3.2 CAP Theorem . . . 12
- 3.3 Data Replication . . . 12
- 3.4 Concurrency Control . . . 13
  - 3.4.1 Two-phase Locking . . . 13
  - 3.4.2 Multiversion Concurrency Control . . . 14

4 Concept 15
- 4.1 Functional Requirements . . . 15
- 4.2 Data Replicas . . . 16
- 4.3 Freshness Metrics . . . 17
- 4.4 Update Propagation . . . 20
  - 4.4.1 Refresh Strategies . . . 20
  - 4.4.2 Refresh Operations . . . 22

4.5 Consistency Constraints . . . 24
4.6 Transaction Handling . . . 25
4.7 Freshness-aware Read Access . . . 26

5 Implementation 28
5.1 Polypheny-DB . . . 28
5.1.1 Placements . . . 29
5.1.2 Query Routing . . . 31
5.2 Lazy Replication . . . 31
5.2.1 Placement Versioning . . . 32
5.2.2 Replication Strategy . . . 33
5.2.3 Replication State . . . 34
5.2.4 Change Data Capture . . . 35
5.2.5 Lazy Replication Engine . . . 36
5.2.6 Automatic Lazy Replication Algorithm . . . 39
5.2.7 Manual Refresh Operations . . . 42
5.2.8 Placement Constraints . . . 42
5.3 Freshness Awareness . . . 43
5.3.1 Evaluation Types . . . 44
5.3.2 Query Specification . . . 44
5.3.3 Freshness Processing . . . 45
5.3.4 Freshness Filtering . . . 47
5.3.5 Freshness Selection . . . 48

6 Evaluation 50
6.1 Goal . . . 50
6.2 Correctness . . . 51
6.3 Benchmarks . . . 51
6.3.1 Evaluation Environment . . . 52
6.3.2 Results . . . 53
6.4 Workload Comparison . . . 67
6.5 Discussion . . . 68

7 Conclusion 69

8 Outlook 71

Bibliography 73

Appendix A PolySQL Syntax - Freshness Extension 78

Declaration on Scientific Integrity 80

1 Introduction

Within the last decades, data has not only grown in volume and variety but also in importance throughout every industry. Whereas data has so far only been considered a mere tool to assist certain tasks, it developed toward inherently driving new technologies and scientific advancements, becoming an industry focus-point on its own [56].
To process the ever-increasing volume and complexity of the data, companies nowadays do not only rely on individual data silos by themselves but utilize all forms of data ingestion methods to extract and connect nested knowledge out of different sources, to gain economical advantages by predicting new trends [19]. However, this also increased the need to reliably store and rapidly access such information, along with constantly varying requirements. Database management systems (DBMS) are therefore progressively challenged, to handle these swiftly growing heterogeneous datasets as efficiently as possible while being able to adapt to new situations.
To meet these new demands and consequently process and extract meaningful information required by data-driven applications, new systems have emerged. To improve the overall data retrieval time, these novel Polystore systems natively combine different physical data engines to adapt to different workloads by leveraging the key benefits of each underlying engine [45, 50]. Although such systems are inherently built to process heterogeneous data with high throughput, they still need to adhere to the given requirements and store all data reliably to protect it against failures. Especially since cloud computing has become a crucial and central part with regards to data processing, companies tend to progressively store and manage their data across different globally distributed data centers [3]. Access to these storages is provided according to the Service Level Agreement (SLA) of the respective provider. Such quality guarantees usually include elastic up- and down-scaling of resources as well as a high degree of availability. This can ultimately be achieved by replicating data throughout different regions, to provide resilient and fault-tolerant architectures [13, 47].
However, to safely manage these large distributed volumes of data, they need to be replicated eagerly to every participating node to ensure global data consistency and avoid losing data. Consequently, this impacts the accessibility of a node and reduces the possibility to be parallelly used for read-operations. Therefore, cloud providers need to design their services to balance between sufficient protection against failures and still providing adequate

response times to query the data [28, 33]. To support varying use cases, data freshness strategies were introduced as an essential part of distributed data management systems.
The freshness essentially corresponds to the age of a specific data item and reflects how current and up-to-date it is. Because they might pursue different goals and recent data is not always equally important for depending applications, they can often tolerate different levels of freshness. Especially for analytical queries, that often spend hours retrieving, extracting, and combining relevant data, slightly outdated data will not drastically impact the final result.
This consequently allows lowering the constraints to replicate data only to a subset of nodes and still efficiently utilizing the remaining resources to be used for data retrieval.

1.1 Polystores

The decision on which structure to use for storing data has a fundamental impact on the overall performance of a system [40]. While row-oriented data stores might be useful and preferred for write-heavy transactional workloads, they are rather insufficient for purely analytical workload which would rather benefit from a column-oriented data store with less write operations [2].
Even though a variety of DBMS exist, which were originally created with an intention to support specific scenarios, applications are getting more complex, influenced by various requirements and characteristics to serve multiple use cases at once. That is why modern-day applications can not solely rely on one storage technology alone. Consequently, Multi- and Polystore systems have emerged. While Multistore database systems aim to combine and manage data across heterogeneous data stores, Polystore systems are essentially based on the idea of combining Multistores with *polyglot persistence* [50]. Polyglot persistence is a term that refers to a practice originating from the concept of *polyglot programming*, to utilize different programming languages for different tasks [27].
Along this paradigm, Polystores want to utilize multiple data storage technologies to fulfill different needs for different application components, in order to cope with mixed and varying workloads. Although, possibly combining various DBMSs, Polystores generally still lack some features of a conventional DBMS. Therefore *PolyDBMSs* have recently been introduced as an expanded specification to regular Polystore systems, extending their abilities and transforming them into full-fledged database systems [52].

1.2 Motivation

Due to the growth in data and complexity, Polystore systems aim to provide fast response times for various use cases and applications of all kinds. These systems natively encompass several different stores, where each is capable of fulfilling a different requirement. This enables an application to harvest the best possible response times for different workloads. Because Polystore systems were originally introduced to support heterogeneous data, they mostly do this by utilizing different stores at once. Since these systems might be distributed, the data could exist redundantly across these stores. For such systems to utilize the different

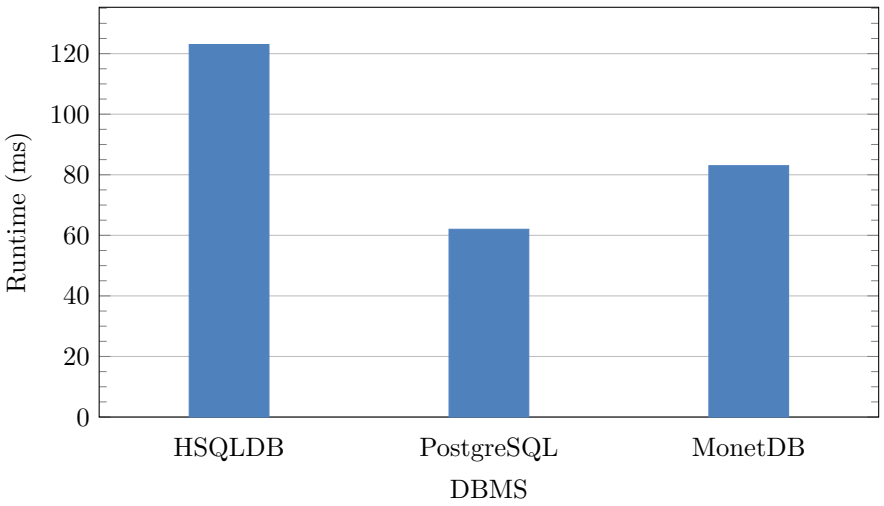


Figure 1.1: Execution Time Comparision of a Single Write-Operation on Different Stores.

underlying engines and consequently improve the read-performance, the data also needs to be consistently written to all relevant stores to leverage their unique benefits. However, this essentially limits the performance of such write-operations to the slowest performing node in such a setup.

As depicted in Figure 6.11, the utilized stores write the same data very differently, leading to large deviations in their execution time. This negatively impacts the overall performance of such systems. Additionally, this might also reduce the availability of the entire system since it might not allow read-operations to be executed in parallel. But because these stores differ in terms of their field of application, we cannot simply remove the slower performing stores to improve write-operations. This would comprehensively neglect all benefits originally introduced by Polystore systems. Essentially, the system needs to be able to allow transactional as well as analytical workloadd to be executed in parallel.

This could be accomplished by decoupling updates on a system, to only target specific stores. These stores can then be used to asynchronously update the remaining stores without directly affecting the user with additional wait situations. Although, reducing the consistency of the system, this approach intuitively generates multiple temporary versions per data object. In order to efficiently utilize the resources of the entire system users or applications can choose to query these versions by considering their *freshness*. Since some requests do not always require the most recent data, the maximum tolerated degree of outdatedness could be specified during retrieval. This specification could then be used to identify a well-suited location to retrieve the desired data object. This notion of freshness aids in efficiently using these temporary outdated versions for reads as well. Hence, it allows the system to execute read- and write-operations in parallel. Further, it mitigates the need to immediately update every existing data replica to the most recent state. Such delayed updates now allow for much higher throughput and increased performance in scenarios where also slightly outdated data is acceptable.

On the basis of Polystore systems, we could now use this scenario to natively leverage the benefit of a heavily write-optimized store to primarily receive any write-operation, which

then distributes the received changes asynchronously to the remaining stores. This would immediately allow, using the notion of freshness to be able to utilize outdated data. Ultimately this loosens the constraints and downsides of Polystore systems to efficiently utilize all available resources.

1.3 Contribution

The contribution of this thesis is fivefold. First, we identify and define the necessary requirements to establish freshness-awareness in general. Second, we outline and propose various possibilities to enable freshness-awareness within a DBMS. Third, we reduce the existing locking constraints and introduce a fault-tolerant replication algorithm, to automatically refresh outdated stores within *Polypheny-DB*[1]. Fourth, we establish a query extension to allow the specification of tolerated and desirable freshness levels on various metrics. Fifth, we adapt the routing system to use those freshness constraints to efficiently identify and route queries towards the appropriate stores, to allow the possibility to decouple primary from secondary transactions.

1.4 Outline

This thesis is structured as follows: In Chapter 2 the foundations and concepts of data freshness characteristics and requirements are presented. Further, we give an overview of the current state of research in the field of data freshness. Chapter 3 visits existing fundamental concepts in the context of distributed data management systems which are necessary for the remainder of this thesis. This is followed by Chapter 4 which describes the functional requirements, necessary to introduce the notion of freshness inside a Polystore system. Additionally, it discusses and proposes possible approaches how to implement them in a Polystore. In Chapter 5 these concepts including all requirements and building blocks will be applied to Polypheny-DB, a specific Polystore system, while Chapter 6 focuses on possibilities on how to ensure correctness and measure the performance of the implementation, including all necessary prerequisites. Finally, Chapter 7 concludes the thesis by summarizing the individual contributions according to the proposed implementations, while Chapter 8 gives an outlook on future work and possible extensions to the presented work.

[1] https://polypheny.org/

2
Related Work

This chapter aims to provide a background for the topics revolving around *Data Freshness* and compares existing approaches as well as related and recent research activities in this field. In the context of this thesis, we consider six topics, which are necessary to establish freshness-awareness inside a DBMS. This Chapter is therefore separated into seven sections, were each part contributes to a characteristic, needed to provide the notion freshness. This includes a definition of Data Freshness in general (→ Section 2.1), possible metrics to define cost functions (→ Section 2.2), which are used in the evaluation process to express and compare different freshness-levels (→ Section 2.3), and how to correctly involve different available versions of data objects (→ Section 2.6). Furthermore, how to identify existing possibilities to replicate data (→ Section 2.4) and propagate updates to outdated nodes in order to refresh them (→ Section 2.5). Finally, this is concluded by a discussion, of how users can specify their tolerated freshness to retrieve data (→ Section 2.7).

2.1 Data Freshness

The consideration of data freshness is a widely used concept among database systems and intuitively introduces the idea of how old or stale data is. Nonetheless, there is no commonly agreed definition, hence several different notions and interpretations exist, on how to characterize and measure data freshness. This was already described by Naumann et al. [35] in 1999, which state that despite these variations, it is strongly related to the concept of accuracy. The accuracy of a data object could therefore be loosely summarized as the percentage of objects without data change. This thought was also shared by Redman who defined the data accuracy as "the degree of agreement between collection of data values and a source agreed to be correct" [42]. Hence, it can be considered as the precision of such data elements in respect to their most up-to-date version.

Peralta [39] also summarized freshness as an implication of how old data is and referred this to a user's expectation on the actuality of an object. Consequently, using the last time objects were updated, to identify any accuracy measures. The author also distinguished between two data quality factors. The *currency factor* expresses how stale data is after it has been extracted and finally been delivered to the user. This concept is often considered

within data warehousing systems, when already extracted materialized data is processed and delivered to the user. Because the extracted source data might have already been updated after it has initially been extracted, consequently resulting in stale data to be delivered. The second factor corresponds to the *timeliness* of data, which essentially states how old data is and captures the gap between the last update and the actual delivery.

2.2 Freshness Analysis and Metrics

A metric is a specific figure which can be used to compare and evaluate given quality factors with each other. Along with the various definitions and approaches to define data freshness itself, the metrics to identify and measure the actual degree of freshness also vary. Several metrics are summarized by [16, 37, 39] and can be categorized as follows:

- **Currency metric**: Measures the elapsed time since the source data changed without being reflected in a materialized view or a replica in distributed setups.
- **Obsolence metric**: Accumulates the number of updates from a source since the data extraction time. For replicated systems, this can be characterized as the number of pending updates that are still waiting to be applied to a secondary system since it has last been refreshed.
- **Freshness-ratio metric**: Identifies the percentage of extracted elements that are indeed up-to-date compared to their current values.
- **Timeliness metric**: Measures the time that has elapsed since the data was updated. Generally defined as the time when the replica was last updated.

Furthermore, [6] and [59] define the notion of freshness by associating it to the *age-of-information* (AoI), which is defined as the timeliness represented by the timestamp of the transactions which has updated the data object. This is enriched further with the *age-of-synchronization* (AoS) that corresponds to the time when the object was actually updated.

2.3 Freshness Constraints

Along the various freshness metrics described in Section 2.2, essentially three types of freshness constraints suggesting a tolerated level of freshness can be summarized [30, 60]:

- **Time-bound constraints**: Are used as timestamps that accept values which have been updated by younger transactions.
- **Value-bound constraints**: Are more commonly used with numbers than with text and consider the percentual deviations from the current value.
- **Drift constraints on multiple data items**: Are especially relevant for transactions that read multiple data items. It considers if possible update timestamps of all required data items are all within the same specified interval. Additionally, for aggregations, as long as a particular aggregate computation is within a tolerable range it can be

accepted. Even if some individual values of the data items are more out of sync than the compound operation.

The notion of time-bound constraints is also shared by the authors of [53]. They propose to measure and specify these freshness constraints with the notion of *Absolute-* and *Delay Freshness* to characterize their proposed freshness functionalities.
Here the *Absolute Freshness* of a data item d is characterized by the commit timestamp t of the most recent update transaction that has modified the item d. These timestamps can be used by the client to individually define their freshness requirements. The younger a timestamp the fresher the data. Additionally, they use *Delay Freshness* to define how old and outdated requested data objects $t(d)$ are, compared to the commit time of the current value $t(d_0)$. Defining their freshness function as:

$$f(d) = \frac{t(d)}{t(d_0)}, with f(d) \in [0, 1] \tag{2.1}$$

resulting in a *freshness index*. Röhm et al. [43] state that such an index consequently reflects how much the data has deviated from its up-to-date version. An index of *1* intuitively means that the data is at its most recent state, while an index of *0* is defined as infinitely outdated. While [57] and [25] both consider the delay-based staleness in the time domain as well, they also consider constraints on an acceptable value-based divergence δ. This aims to measure the difference between two numerical values by analyzing their similarity on basis of their absolute value $|d_0 - d| < \delta$, where low values between the base and replicated data reflect a more up-to-date replica for a given object.

2.4 Update Propagation

An essential part to gain performance with freshness-aware data management is loosening the update situations by reducing the number of replicas that need to be updated immediately. This decoupling between necessary updates on a primary site and deferred updates towards secondaries reduces the total update time and in contrast, increases the overall availability of the database system [32]. This mechanism itself suggests to the user that the database system is updated eagerly while internally the updates are actually propagated lazily. However, this has the side effect that the secondaries hold slightly outdated or stale data. Due to the decoupling, these can now efficiently be utilized as read-only copies to speed up Online Analytical Processing (OLAP) workload while still performing Online Transaction Processing (OLTP) workload at the primary level as [41, 43, 57] have pointed out in their work.
Depending on the given system architecture, different replication-strategies have been proposed on how updates are propagated towards outdated nodes. Wei et al. [54] propose to replicate data in form of an update policy adaptation algorithm, that dynamically selects update policies for derived data items. Which are then executed based on internal system conditions. These conditions are defined as several layers of validation a transaction has to pass. E.g. in the validation stage, the system checks the freshness of the accessed data. If it is considered to be not fresh enough, the entire transaction will be restarted. This

imposes huge performance mitigation and wastes resources since the transaction itself could be processed and re-queued multiple times before it might actually get executed.
Psaroudakis et al. [41] mention the existing design gaps between OLTP- and OLAP-oriented database management systems and describe the importance of allowing workloads to be executed in parallel. However, they solely focus on a single table-level rather than a replication scenario in a distributed environment. Furthermore, they are interested in real-time processing and claim that even the slightest outdated data is unacceptable to provide real-time reporting. To fulfill these mixed workload requirements they summarized the idea of *SAP HANA*[2], which offers a main area that contains the base data and a delta area, which supports transactional operations and includes recently changed data. Read operations, therefore, need to query both main and delta area jointly to provide results. Since the delta area could increase without bound, it is periodically merged into the main section.
A similar approach is shared in [54] which tried to mitigate the complexity and replication overhead by combining base data elements with derived elements. For queries, a base set of information is enriched with delta fragments to derive the relevant output. Since the base information is always available and queued changes are applied during runtime to recompute the actual response, this will reduce the needed update cycles on all replicas. Nonetheless, this approach has a higher cost when encompassing several derived data sets, and is therefore rather suited for values that can easily be derived.
To increase the transactional throughput Pacitti et al. [38] introduce concurrent replica refresh operations. They discuss the idea of a *Preventive Replication* algorithm by using an asynchronous primary-copy replication approach, while still being able to enforce strong consistency. This is achieved by utilizing a First-In First-Out (FIFO) queue to correctly serialize any updates which shall be delivered to secondary replicas.
The authors in [53] propose several multi-tier layers to handle replication and freshness scores. Nodes are classified into two categories, either as updatable or read-only, and placed onto three levels. Level-1 nodes receive all updates immediately acting as the primary servers, level-2 nodes are read-only nodes containing as up-to-date data as possible while the level-3 read-only nodes are updated rather infrequently. These layered levels are composed as a tree, receiving asynchronous updates from the lower level. In this case the higher the level, the more outdated the data.
In contrast, [57] approaches its replication pursuit by establishing a relationship between individual data items, represented as a Directed Acyclic Graph (DAG). Their graph essentially denotes a dependency between data items. The root node of such graphs corresponds to a base data item and child nodes correspond to its deviations over time, which as well corresponds to a multi-layer replication setup.

2.5 Refresh Strategies

Based on the lazy propagation algorithms presented in Section 2.4, three refresh and different propagation strategies could be identified when updates should be propagated to replicas in

[2] https://www.sap.com/products/hana.html

order to refresh them.
Despite that the authors of [41] rather focused on table-level objects and not on comprehensive system replication scenarios, the update mechanism still follows the same requirements. To jointly support a mixed workload of OLTP- and OLAP-oriented transactions, it suggests a periodic approach to merge the main and delta areas. This essentially refers to updating an outdated base item, in this case, represented as the main area.
Röhm et al. [43] avoid scheduling such update propagations entirely. They suggest to simply decouple the primary update transactions from the read-only nodes and immediately executing the propagation afterward. Although this aids the throughput of the initial update transactions, the secondaries stay outdated for almost no time at all, leaving no room for possible outdated replicas with different versions to even exit. Furthermore, such approaches as well as periodic scheduling completely neglect to verify the current load on the underlying system. Since this could unnecessarily cause performance mitigations on the replicas to be updated, it should always be accompanied by identifying the idle time of replicas, as well as ensuring that the current load on the replicas is within bound and is, therefore, able to endure such an update [53].
In contrast [39], determined the goal to identify the minimum set of refresh transactions, such that it is guaranteed that a node contains sufficiently fresh data concerning the users requirements. Therefore, any outdated node will independently pull the number of updates that are necessary to fulfill future requests along with the requirements.
Wei et al. [54] follow a different approach by analyzing the *Access Update Ratio* (AUR) for any data object *d*.

$$AUR(d) := \frac{AccessFrequency(d)}{UpdateFrequency(d)} \tag{2.2}$$

As depicted above, any item beyond an AUR of 1 is considered to be accessed at least as frequently as it is updated. Hence, it is considered as *hot* and shall be updated as soon as possible for applications to receive the real-time value, whereas a ratio below this threshold refers to *cold* data and establishes that this item is accessed comparably infrequently and thus needs no immediate refresh.

2.6 Consistency

There are multiple constraints to enforce when trying to ensure consistency in a distributed setup. The authors of [54, 57] discussed data freshness together with the scheduling of update transactions from the point of view of real-time applications. They elaborated the concept of temporal validity with respect to value-based freshness, such that specific values are only considered valid for a certain time interval before becoming outdated. Hence, they describe the necessity to classify objects into temporal data that can continuously change to reflect a real-world state and non-temporal objects that will not become outdated over time like the validity of an ID card. This concept essentially pursues the fundamentals of temporal databases, which keep historic values as well as their validity-intervals, to allow the reconstruction of any past value [22].
Voicu et al. [53] proposed to decouple transactions in order to correctly separate individual

requirements for update propagations. These transactions are differentiated into four variations. Firstly, regular update transactions that target primary nodes, secondly propagation transactions that refresh read-only nodes, followed by refresh transactions that are executed if a freshness level on a local store cannot be provided and finally, OLAP related read-only transactions. Furthermore, they require that data accessed within a transaction is consistent, and independent of its freshness. To correctly serialize the updates they ensure that their update serialization order is the commit order of the initial update transactions.
The authors of [41] introduced a common data structure to provide access for both OLTP and OLAP. They enabled the usage of delta and main storage together with the utilization of multi-version concurrency control (MVCC) to allow both workloads to be executed in parallel. This implicitly enables the system to keep several versions of a data item (see Section 3.4). These versions can be directly used as outdated replicas or to provide certain freshness guarantees. Although this would indeed improve update times and would support referential integrity, the implementation of MVCC is complex and needs more system resources than usual.
Finally, the authors of [5] propose freshness locks that are applied on stores that have been selected to fulfill the read requests. These locks aim to provide fast response times for analytical workload and essentially ensure that data is not refreshed as long as it is being read by another transaction.

2.7 Freshness-aware Read Access

The fundamental idea of data freshness aims to utilize all provided resources to improve the read access while transactional workload is being executed. Therefore the read access is a matter of efficiently routing any client request, to a suitable replica in respect of the required level of freshness. According to [5, 43], a router needs to utilize system state information on replicas to identify the best way to route a query.
In [53], clients are able to contact any read-only replica directly. For every read-only transaction, they can as well specify their tolerated freshness, by providing a timestamp which is internally converted to a freshness-index. If the replica is able to fulfill the request it directly returns the response to the client. If it currently does not possess a sufficient freshness-level, it routes the request to another node, which can be identified by routing it towards the root of the tree. The parent is then able to identify which node is capable to fulfil the condition. If none of the replicas can execute the request within this subtree, a refresh transaction is triggered, which will update and refresh all nodes before processing the read operation.
The authors [31] introduce a central *preference manager as a service* at the client site. This manager can suggest where to route queries based on given cost metrics. By comparing individual latencies for any request, they aim to improve the overall performance by deliberately choosing if a request shall be routed to a primary or secondary site. This analysis is influenced by the *Replication Lag* inside a MongoDB cluster which refers to the average time that passes before an update is propagated to the secondaries.

3

Foundations of Distributed Data Management

This chapter describes concepts and general foundations, which are necessary to supplement the remaining content of this thesis. These foundations are mainly associated with topics on distributed data management and the different challenges that need to be considered.

3.1 Data Partitioning

Besides the utilization of the correct storage engine, the used data model and structure will also have an impact on the overall performance. Depending on the query or how data is accessed, data partitioning can be used to increase the efficiency and maintainability of the system [4].
The process of data partitioning refers to splitting the data into logically and sometimes even physically separated fragments that can be used independently.
In general, data partitioning can be distinguished into two variations. Since both of these split the data into multiple parts, it is often referred to as data fragmentation [60].

Vertical Partitioning is usually applied at design-time of a data model inside a database. It involves the creation of tables with fewer columns and therefore using additional tables to store the remainder of columns [18, 36]. In order to combine and reconstruct these vertical partitions again, there needs to be a logical reference, which assists in uniquely identifying individual items.

Horizontal Partitioning refers to the partitioning of objects like tables into a disjoint set of rows. These benefit from being stored and accessed separately [15]. To support this rather explicit form of partitioning, there exist various partition algorithms. These algorithms can be functionally applied to a table, based on an arbitrary column. This results in a fragmentation of the table on the basis of the data values of the selected column.

Data partitioning generally enables a system to process data concurrently and to some extent even in parallel, considering that access to data can be efficiently load-balanced and therefore enhances the throughput per query.

Although data partitioning is often associated with the improvement of query performance, it can be also be used to simplify the operation of a database cluster and therefore help to increase the overall availability. Through the replication of partition fragments, the data resilience of the system can be improved. Even if part of the data storage nodes are temporarily not reachable, your system still might be fully operational and available due to the replication and distribution of data fragments [14].

3.2 CAP Theorem

An essential part of distributed systems is the handling of failures or outages of participating components. The *CAP theorem* [28], introduced by E. Brewer [12], discusses these scenarios and states, that it is not possible to keep the system available while providing global data consistency at the same time. This problem is driven by the claim that a node within a clustered system, cannot identify whether another node or the network connection in between has failed (network partitioning).
Although this theorem was primarily introduced to support the differentiation between *Availability* and *Consistency*, it only formulates the trade-off in cases of failure. Since this should rather be the exception, Abadi [1] generalized the concepts of Brewer by introducing *PACELC* as an extension to the CAP theorem. This essentially adds the more common non-failure case to the definition. He claims that it is not sufficient to reduce the decision on the occurrence of the failure alone. Because even in high-available environments the data needs to be replicated to ensure availability and thus latency. Therefore, the possibility of failure alone, even in the absence of a failure, implies that the availability depends to some degree also on the data replication.

3.3 Data Replication

In distributed setups, the utilization of data replication is crucial to improve the query performance by replicating certain partitions of data to where it is needed [58]. Furthermore, replication also increases the availability of the system and protects it against outages or performance mitigations, by allowing workload redirection and load balancing to healthy replicas in the cluster [32]. However, maintaining the consistency of all available replicas results in scalability problems. Abadi summarizes “three alternatives for implementing data replication” [1] in a distributed setup. He states that these different nuances inherently result in the aforementioned trade-offs between availability and consistency described in Section 3.2. The system can either choose to send the updates to all replicas at once, send updates to a predefined replica first, or to a single arbitrary node.
While the first approach can be directly applied to a system, the second and the third require the node that has received the initial modification to trigger an additional update operation. Generally, the data replication approaches can be ultimately distinguished into two approaches [29]. *Eager Replication* provides a strong consistency among all replicas. Each modification will first be applied on all nodes before the update transaction is considered to be successful. Hence, no stale data retrieval is possible, and users can choose to

query any of the available nodes. Because the update is however applied synchronously, the transaction blocks until the last replica has finished the write-operation. Since this is done within one global transaction, specifically in heterogeneous environments, the performance of an update is bound by the slowest performing node.
In contrast to its strict counterpart, *Lazy Replication* decouples the primary update transaction from the update propagation to secondary nodes. In its basic form, it only supports weak consistency. Since updates only need to be acknowledged and executed by one store, the update propagation to the remaining nodes is executed asynchronously [25]. This improves parallel processing and increases the availability because only one node is blocked during the update. All other nodes can still serve incoming requests, which especially increases the popularity of analytical applications [20]. However, during the convergence period, until all changes have been replicated, the system is exposed to an inconsistent state. As pointed out by [16], utilizing a lazy propagation of updates, immediately leads to different versions of participating data items and thus also provides stale data. This could result in retrieving outdated data if the client contacts one of the outdated nodes.
Since the initial transaction defers the update propagation, this approach automatically results in *Eventual Consistency* [44]. Although not considered strong, this form of weak consistency guarantees, that if no further updates are made during convergence, all accesses to these replicas will eventually see the same value and become uptodate [32].

3.4 Concurrency Control

A major topic in the context of distributed database systems is *concurrency control* [8]. It is directly associated with the *Isolation* criteria of the transactional *ACID* properties. With associated locks of data items, it ensures the correct results and treatment of interleaving operations in a database system [11].
As illustrated by the authors of [55], not solely the data replication will impact the trade-offs between latency and consistency, but so does the concurrency control.
Despite the fact that there exist multiple protocols that handle concurrency control differently we will discuss two of the most common ones: *Two-phase Locking* (2PL) and *Multiversion Concurrency Control* (MVCC).

3.4.1 Two-phase Locking

As the name suggests, the protocol itself is divided into two distinct phases. An *expansion phase*, where locks on required objects are gradually acquired, and no locks are released, and a *shrinking phase* where locks are released, and no locks are acquired anymore. In summary, no lock can be acquired as soon as some locks have already been released.
The basic approach differentiates between exclusive locks, which can only be acquired by a single transaction at a time, and shared locks that can be acquired by multiple transactions. While write modifications lead to an acquisition of a new lock, reads can attach itself to an existing shared lock instead of acquiring a new lock [34].
This however can lead to the starvation of write operations, since shared locks can be more

easily extended with new transactions, leaving write transactions on wait until there are no more transactions in the list of the shared lock.
Although this protocol is most commonly referred to as 2PL, it provides additional extensions and variants that are used in practice. These variations only differ in the time when the second phase starts releasing the locks. While *2PL* releases locks once the operation has finished, this can lead to dirty reads [55] since the transaction has not been committed yet and could still be rolled back. The more rigid case *strict two-phase locking* (S2PL) only released the locks of write-operation as soon as the transaction ends. Shared-locks on the other hand are released early. The most rigorous version is the *strong strict two-phase locking* (SS2PL) which releases all locks when the transaction has ended. This is either at commit time or when the transaction is aborted. Although, this form of 2PL is effective to achieve distributed and global serializability, and provides automatic deadlock detection and resolution. Due to its rather pessimistic blocking behavior, it negatively impacts the performance of the system.

3.4.2 Multiversion Concurrency Control

While *SS2PL* suffers from lock contentions between long-running analytical reads and update transactions, it cannot support parallel writes and reads on the same object. To solve this problem *MVCC* [8] was introduced, by keeping multiple data copies per object and generally omitting locks. It was originally established for multi-version databases that tried to extend the concept of shadow pages, by keeping the complete history or at least multiple versions of each object [9, 10]. The idea is that every new write of an object x, by transaction T^k creates a new version x_k for this object x.
Most commonly, MVCC provides a *snapshot isolation*, which was introduced by Berenson et al. [7]. This snapshot enables each transaction to see the state of the data at the time when the transaction has started. While this allows to compare and use different versions that are already considered to be outdated [23], hence providing more flexibility for concurring transactions. It is more difficult to find a corresponding non-conflicting execution that is equal on all stores within a distributed setup [20, 26].
Moreover, MVCC protocols are challenged with the decision of how many versions to retain and when old ones become obsolete and can be removed. Generally, this leads to a larger data footprint since objects are stored redundantly, natively increasing the size of the system.

4
Concept

In this chapter, we will define all functional requirements necessary to establish freshness-aware data management in a PolyDBMS, as an enhanced Polystore system. Consequently, we will discuss the contents of Chapter 2 and propose solutions how these approaches and techniques can then be applied to a PolyDBMS. Hence, this chapter is separated into several sections where each represents a necessary building block to provide the notion of data freshness.
Since we will use PolyDBMSs as an extension to Polystore systems, we will use the term Polystore and PolyDBMS interchangeably.

4.1 Functional Requirements

Based on the provided discussion in Chapter 2 we have summarized and defined six fundamental functional requirements, which are necessary to generally provide the notion of freshness within a system. These prerequisites are however not unique to Polystore systems and can be applied to any database system.

(i) *Versioning* - The existence of multiple versions per data object is necessary to distribute the workload and reduce the overall latency.

(ii) *Metrics* - Freshness metrics are needed when comparing different versions against each other. They are used to define the outdatedness per replica and help to analyze how much it deviates from an up-to-date version.

(iii) *Data Replication* - Data needs to be correctly replicated across the system to refresh a specific version. All replicas should therefore be able to be updated independently.

(iv) *Version Consistency* - The consistency of the participating stores is ensured. Regardless of the role, each version always needs to be consistent in regards to its primary. Even if the state of the versions defers, a given version must always be equivalent to the version the primary had at this time.

(v) *Freshness-aware Read-operation* - The tolerated level of freshness needs to be expressed and specified to retrieve relevant information.

(vi) *Isolate Freshness* - Freshness-related operations should not directly impact or interfere with the system. We need additional transactional semantics to shield them against regular operations.

Since there might be several possible approaches to fulfill the requirements listed above, we will define and conceptualize several possibilities in the subsequent sections, tailored to be solved by a PolyDBMS.
Except for the obvious existence of multiple replicas per data object (→ Section 4.2), there are several prerequisites and requirements to establish freshness-awareness. We essentially have to consider how to express freshness and find suitable metrics to measure it (→ Section 4.3) and further provide users with a possibility, to formulate an acceptable level of freshness (→ Section 4.7). Based on these fundamentals we need to consider how update transactions can be decoupled and defer the refresh of specific replicas (→ Section 4.4), all while ensuring the consistency of the entire system (→ Section 4.5) to finally improve the query routing to identify freshness levels to increase the performance of read-only operations.

4.2 Data Replicas

One of the main requirements of data freshness is the necessity and existence of different versions that can receive outdated data. Only with these versions, we are even able to compare their state and define freshness for a given replica. However, it should not be generally required for every data object having multiple versions, but is necessary as soon as you want to consider it in the context of freshness.
As already discussed in Section 2.5 it is crucial to define and assign roles to loosen locking situations on nodes and minimize update times overall. The related work clearly distinguished between primary-update nodes and read-only nodes, where read-only nodes cannot be directly updated by the user and will only receive refresh updates internally from the primary nodes.
Since PolyDBMSs inherently replicate or distribute their data across multiple stores, we already have multiple replicas containing the data. Because we aim to reduce locking situations by decoupling primary updates from secondary transactions, we can simply use a lazy replication approach to propagate the updates asynchronously. By definition, this solution automatically creates multiple versions by deferring the update propagation to secondaries. This immediately leverages the nature of Polystore systems in such a way that no additional replicas have to be artificially created to support multiple versions for each data object *(i)*, which will eventually converge. The respective replication approaches are discussed in more detail in Section 4.4.2.
Since we always need a foundation for all freshness-related comparisons, we will need at least one up-to-date replica. This replica should contain relevant information to illustrate the deviation from another possibly outdated version on a different node. To achieve this in a Polystore environment we need to be able to classify existing stores into specific groups or roles. Naively these could be *up-to-date* and *outdated*, where the latter could become outdated over time, while the first one always needs to be up-to-date.
Based on these roles, we are then able to decide which replicas to consider for which use

case. Alongside the idea of a Polystore system, where each underlying engine has its own purpose, we can directly apply these roles based on the provided use case. We could e.g. label those stores as up-to-date that support highly transactional workload and will therefore be considered for every update as OLTP nodes, and configure replicas as outdated, when they are rather suitable for analytical queries which implicitly harvests the benefits of the encompassed stores as OLAP nodes.
To consequently compare those replicas, they need to be equipped with metadata of at least two timestamps. One is the update timestamp when the replica has been last modified, the other is the commit timestamp of the original transaction which has modified the replica. This differentiation is important since it allows us to compare replicas based on their commit timestamp. This timestamp is always directly associated with the primary transaction. It means that even if the update propagation for secondaries has been deferred to a later point in time and consequently has a different update timestamp, as soon as they converge they will have the same commit timestamp as their primary counterpart. Otherwise, it would not be possible to compare replicas based on their timestamp, since individual commit timestamps per replica would not allow a direct timestamp comparison to determine the freshness of an object.

4.3 Freshness Metrics

As discussed in Section 2.3, freshness can be considered with several indications and nuances. There is no unified definition of data freshness or common freshness metrics. These rather depend on specific use cases and system requirements. While freshness extractions on value-based divergence are only really suitable for numerical values, to measure the deviation from a base item, time-based freshness on the other hand can be applied to arbitrary data types and can therefore be used in a more general notion. Because the perception of freshness is rather subjective and depends on the use case, the time-based constraints are often still not sufficient for very frequently updated replicas. The accuracy would differ greatly when one node has received an update within the last minute and might be considered fresh, but this one particular update might have changed the entire table. This would then rather reduce the freshness to a simple version-comparison and allow questions such as, *"if it exists, how did the data item look like roughly one minute ago"*. Admitting that this might be desirable for some use cases we want to extend this notion by considering deviations from the primary copy as well.
Hence, we propose that users can specify their tolerated level of freshness in a variety of ways. All proposed metrics will apply filters based on the abstract equation in 4.1, This essentially validates per replica whether it can be used for the tolerated freshness-level δ regarding a given data object d, where δ can be associated with any metric described in the following definitions.

$$F(d, \delta) := \begin{cases} \text{true} & \text{if } f(d) \geq \delta \\ \text{false} & \text{otherwise} \end{cases} \tag{4.1}$$

Given that the freshness filter function $F(d, \delta)$ is valid for all described metrics, the concrete

freshness determination is individually defined as $f(d)$ varies among the use cases. In general, this function is defined to return a specific level of freshness for a particular data object d, where the object d is available on all replicas and can vary in terms of freshness, due to different update times. While the individual freshness function returns a calculated freshness, the filter function will remove any replica that does not meet the designated freshness-level δ. We only require that δ and the return value of $f(d)$ are of comparable types.

Absolute Timestamp A timestamp can be directly specified as a lower bound threshold, during replica comparison. Identifying all replicas that have been updated more recently than the specified timestamp, to be considered during the selection process. The greater a timestamp, the younger it is, respectively also the higher a timestamp is, the fresher it is. With this function, the freshness of a data item d is directly returned as the commit timestamp when this object has been written. As mentioned in Section 4.2, the commit timestamp $t(d)$ can be referred to as the current state of this replica, and is associated with the commit time of the transaction within this operation that has been applied to the corresponding primary copy.

$$f(d) := t(d) \tag{4.2}$$

In this case, δ is a user specified timestamp $t_{timestamp}$ and consequently compared against $t(d)$ to verify if the freshness-constraints are met.

Absolute Time Delay Any delay can be useful to intuitively specify the accepted level of freshness, without explicitly specifying a timestamp as a lower bound. This function rather allows specifying a time delay based on the current time t_{now}. This metric, therefore, allows specifying freshness with respect to the current time, resulting in recently updated replicas considered to be fresher than others. Although not directly specified as a timestamp, an absolute time delay will still generate a timestamp for comparison to be used in δ. The delay is simply subtracted from t_{now} to again generate δ as a lower bound timestamp used for comparison. The freshness evaluation is therefore equal to the equation 4.2 and provides a different approach to specify the tolerated freshness. Both construct a timestamp that acts as a lower bound for acceptable replicas. This can be used as a filter to check for each candidate if it is fresher or respectively has received a state where its commit time is newer than the lower bound. If not, the replica is removed from the list of possible candidates.

Relative Time Delay Although, an absolute time delay is useful in some cases by defining its freshness based on the current point in time, it might lose some detail and could filter some replicas that in some scenarios are actually useful. If e.g. an object has not been modified for a few hours and even if the replicas might already be up-to-date, they will not be considered when specifying an absolute time freshness that accepts the last hour. This is also true if the secondary replica is not up-to-date yet and is still in the process of convergence. Disregarding its state, it will be avoided since its current commit timestamp is out of bound of the specified time delay. Even though

intuitively these replicas are considered rather fresh in respect to their primary copy. Therefore, if we merely want to observe how much a secondary might deviate from the primary in terms of the update timestamp we need a new metric. We consequently need to specify the accepted level of freshness based on the divergence from its eagerly replicated counterpart, and for that reason provide a relative time delay used during comparison. With this metric the specified relative time delay can directly be used as δ. The freshness function $f(d)$ described in 4.3 will essentially compare the current commit timestamp $t(d)$ against its primary replica $t(d_{primary})$.

$$f(d) := t(d_{primary}) - t(d) \tag{4.3}$$

Again if the calculated deviation from the up-to-date node is within the bound of the specified delay in *delta*, this replica is accepted.

Replica Deviation Although the first three metrics already provide some granularity to consider different nuances of freshness, we do not yet involve the number of pending updates to any replica or can differ between the number of modifications each replica has received yet. For objects with a comparably high update frequency, the notion of timeliness can hardly be utilized to assume the freshness. Therefore, we again want to provide another possibility to allow the specification, based on the divergence between primary and secondary.

This freshness can be specified by a freshness index as proposed by [43]. This ratio can be evaluated and consequently generated based on the number of modifications the primary and secondary copies deviate from one another. Where $m(d)$ is defined by the number of modifications a data object d has ultimately received on a given replica and $m(d_{primary})$ as the corresponding up-to-date copy to compare against. Although a freshness index does not intuitively provide an observable threshold at first glance, it indicates how accurate a given replica d is with respect to the number of modifications of an up-to-date version $d_{primary}$.

This **Modification Deviation** is defined in 4.4.

$$f(d) := \frac{m(d)}{m(d_{primary})}, with f(d) \in [0, 1] \tag{4.4}$$

Generally, this describes how far behind an outdated replica is, compared to the primary version. It can also be used when multiple tables are joined. We can then sum the joint number of modifications and compare it against the current number of update transactions, to give a joint accumulation.

Because it also might be desirable to specify a freshness index but to consider a time deviation as described in [30, 53], the freshness function can be adjusted as needed to also compare the effective commit timestamps as a **Commit Time Deviation**. Since we ensure that the commit timestamp of a primary replica will never be greater than its eager counterpart. We can define the time deviations as an index that is generated with $t(d_{primary})$ being the commit timestamp of the up-to-date replica and $t(d)$ the timestamp of the possibly outdated replica.

$$f(d) := \frac{t(d)}{t(d_{primary})}, with f(d) \in [0,1] \tag{4.5}$$

All the mentioned freshness metrics can be used to compare different replicas that contain a data object d and filter them based on the provided function $F(d, \delta)$. This allows to specify the tolerated level of freshness δ from a type $\tau \in \{TIMESTAMP, TIME-DELAY, INDEX\}$ to be used within the query specification.
Although, as mentioned in Section 4.2, some engines within a Polystore might be more suitable for up-to-date replicas than others, we do not limit the possibility of different freshness metrics to a subset of stores. Hence, every store can uniformly work with all levels of freshness. The Data Freshness shall always be evaluated within the Polystore layer and is then used to compare the tolerated value against possible candidates that might fulfill such a request (see Section 2.7). This enables us to use a Polystore to fulfill the requirements of *(ii)*, by centrally analyzing the freshness constraints and selecting possibly outdated candidates that conform to the specified constraints.

4.4 Update Propagation

To generally allow a system to handle transactional and analytical workload in parallel, we need to reduce occurring locking situations, such that write and read operations do not drastically interfere with each other. Since a Polystore system acts as an abstraction layer on top of the encompassed stores, we can leverage it to act as a coordination service allowing us to restrict the eager replication and locking mechanisms to the primary nodes alone. Given the multiple versions described in Section 4.2, we could consequently decouple primary transactions from secondary transactions. In that sense, any modification to an object will now only target and lock its primary copies which are labeled as *up-to-date*. The secondary nodes could then be read without any further locking by a user transaction. Nonetheless, we need an approach, how these outdated stores can converge their state towards the state of their primary copy. Otherwise, we would have entirely outdated stores that would remain in their current state, and users querying these stores will always obtain stale data.

4.4.1 Refresh Strategies

Since we have no longer access to an eager replication, we need the possibility to apply the changes lazily to any outdated replica. Depending on the use case there might be different approaches needed to fulfill certain requirements. We, therefore, propose the following strategies to apply pending updates to the outdated nodes:

Immediate Execution This approach mainly pursues the decoupling of one single eagerly applied transaction, into two subsequent transactions. While the eagerly applied modification is executed synchronously it is bound by the slowest performing node in a setup. Consequently, with a lazy update propagation, we only have to wait for the up-to-date replicas to finish the transaction. Since these can be strategically placed on stores that are suitable for transactional workload, they are assumed to apply the

operation faster. After this primary transaction has then committed and the locks are released, an asynchronously executed secondary update transaction can be executed, applying the updates to the outdated nodes.
Since these secondary transactions are merely executed with a small processing delay, assuming that the update queue might not be very large, and updates can be applied right away, the outdated nodes intuitively will not deviate much from their primary partner.

On-demand Refresh Since a regular queue will not consider priorities, and we still have to obey the execution constraints of the primary transaction (see Section 4.5) we always need to preserve the execution order of the primary transaction. Depending on the size of the update propagation-queue some replicas might stay outdated longer than others. For this reason, an on-demand approach is necessary to refresh outdated nodes at once and bring them up-to-date. To avoid that updates are not applied twice, executing such an approach will remove all pending updates for that specific replica from the queue.

Load-aware Although an automatically scheduled and a manual execution will serve most use cases, it might not be desirable to do so. Neither an immediate execution after a primary transaction nor an on-demand triggered refresh, take the system load into account. Since PolyDBMSs can consist of several potentially heterogeneous stores, they might also differ in terms of their computing resources. So while most of the underlying stores could apply the pending changes immediately, some might currently not be capable on handling the additional load and have to be deferred yet again. Despite that the approach might slow down the convergence speed of the updated nodes, it will observe underlying stores and artificially limit the load on the system, introduced through the propagation of updates hence keeping the system stable and available.

Update on Read So far, all proposed solutions suffer either from additional evaluation overhead or from manual interference. This will limit the overall performance of the system. We could accordingly introduce a decoupled operation that is automatically triggered as soon as an outdated replica has been part of any query. During the freshness-related retrieval of any data object, we identify that this is indeed an outdated node and can directly schedule an update propagation for it. This propagation is then executed asynchronously after the initial read-operation has finished. This would reduce the additional caching, and avoid storing information as to which updates need to be applied on which node.
However, the downside of this approach is, in highly transactional environments with heavy write- and read-operations, the outdated node would always be marked as needing an update. This would schedule an update, although it might have already been scheduled by another strategy. To mitigate this, the strategy could be enhanced even further by allowing a centrally defined configuration threshold, that validates how much an outdated replica deviates from its primary. This assumes that if an outdated

replica has been read and is above the centrally configured threshold, no update propagation will be automatically scheduled. This will avoid the permanent scheduling of an update for every freshness-related read access.

4.4.2 Refresh Operations

The Update Propagation generally refers to the refresh operation that transforms possibly outdated objects towards an up-to-date state. Disregarding the described Refresh Strategies from Section 4.4.1, we need to converge the outdated replicas towards their primary copies. There are several possibilities to achieve this and a system can choose to implement each of these cases in various ways. However, each implementation comes with its own trade-offs. We have several possibilities on how to handle and propagate the updates. Since we assume that every write-operation needs to go through the Polystore-layer, we can easily keep track which operations have been applied to the primary node. To ensure the overall consistency, we require that the operations are executed in correct execution order and therefore need to apply all pending changes as they have been applied at the primary site. This imposes a natural execution order of any item in the queue to be delivered to the secondaries.
We need to define how executed operations are tracked and how they then apply those operations to the replicas. Therefore, we propose the following approaches:

Change Data Capture As the name suggests the *Change Data Capture* (CDC) approach aims to preserve every modification that has been applied to the primary node, cache it, and ultimately apply it to all relevant outdated nodes. The idea of this approach is that all changes including the data could be temporarily stored either in-memory or persisted onto a disk. The choice of where to store it depends on the individual consistency-availability requirements and will not be part of this discussion.

For the CDC-algorithm, we consider that during an active transaction all changes will be tracked and written into a FIFO queue. Since this is only a preliminary step, we will conveniently define this capture-collection a *capture-queue*. As operations are being executed, each change along with its data and the corresponding parent transaction is stored within this capture-queue. Although we could simply capture the executed statement in a Write-Ahead-Log (WAL) and re-execute it on the underlying stores, we now benefit from the Polystore layer. Since every query has to centrally pass the Polystore layer to be executed, it will pre-compute and evaluate certain functions or constraints internally at runtime instead of delegating it to an underlying store. Therefore, we can ensure that the values received by a store are equal on all other stores as well. Thus, we can save further computations and store the end result that is pushed-down directly to the designated stores. As soon as this transaction has been successfully committed, all entries in this capture-queue are further enriched with the respective commit timestamp of their parent transaction. Afterward, the corresponding entries in the capture-queue are added to an actual central replication queue containing the pending updates. For each designated replica, that should receive this update an individual entry is created inside this queue. Each entry is accompanied by

its parent transactions ID, the commit timestamp as well as the data to be replicated. Since we do not need to store the data n-times for n replicas determined to receive the data, we can simply link each replication item in the queue to its corresponding replication data, which is stored separately. Since all entries in this final replication-queue are ordered with respect to the execution order of the original transaction, we have ensured that the operations are executed in order to converge to the same state as its primary copy.

Finally, if the transaction aborts, all active entries in the initial capture-queue can be removed due to their association with the parent transaction.

Primary Data Snapshot Although CDC will correctly recreate any secondary replica, it will lose its efficiency when there are almost as many modifications to apply to secondaries that there have been totally applied at the primary site. Even though it would still produce the correct result, it could be further optimized without replicating operation by operation until the replica has converged.

Therefore, another proposition could be the usage of a primary-copy approach. Intuitively this would allow to simply snapshot the entire state of a matching primary node to be copied onto the target replica. During this copy, we only need the current commit timestamp of the primary and snapshot the current state of the respective data object. This could be done simply by executing a read-only transaction to retrieve the current state of the primary replica. Since the snapshot itself will have no real impact on the primary node, we can continue to use it for all operations. Because the secondary replica will be recreated from scratch, querying it will result in an incorrect state. Therefore, it cannot be actively used for any freshness-related queries. Hence, we have to refrain from providing this replica as a possible candidate in the retrieval process and lock it entirely until everything has been processed and the replica is equal to the snapshot. After it has been applied, we can now update the commit timestamp of the replica with the timestamp retrieved alongside the snapshot, to mark this refresh as successful.

Despite that this snapshot-copy will again result in a correctly updated secondary node, it is not suitable for very large data sets to copy. For one, depending on the refresh strategy proposed in Section 4.4.1, it could be triggered too frequently and would constantly lock the secondaries. Additionally, a complete copy of a data set takes time, which removes the replica from the potential candidate replicas to be used within retrieval.

To avoid these locking situations, we could further adapt this algorithm to create a temporary shadow replica while the copy process is in place. With this, we could recreate an entirely new replica based on the snapshot, which would still allow accessing the old outdated node. Although possibly more outdated data is now retrieved, and the data footprint is temporarily increased, this replica can still continue to serve freshness-queries since it will not need to be locked. Finally, when the process has finished, we only need to apply a lock during takeover time to ensure consistency. During this short timeframe, the old replica is dropped and the newly created hidden

shadow replica is now activated, making it an official replica to be used.

View Materialization Along with the idea of the *Primary Data Snapshot*, the materialization of views could also help to reduce the number of statements necessary to create new levels of freshness. Since materialized views are by nature considered to be precomputed-snapshots of data objects, we can simply leverage these semantics to create different versions of data, represented by individual views. Because views are common in most databases, there are an easy to use access without implementing an entire refresh algorithm.

In contrast to the benefits described in Section 4.2, we now indeed need to artificially create new versions. However, instead of replicating the data operation-wise to another store, we can simply omit creating replicas that become outdated and create materialized views on these stores instead. Hence, we are left with at least one true up-to-date replica and several outdated replicas, which are represented by views created on the underlying stores. Due to their flexibility, we can decide per use case which degree of freshness a view supports. Analogously to the aforementioned approach, anytime a propagation or refresh operation is being executed, a new materialized view is generated on basis of the up-to-date replica. This also omits replicating single operations entirely, hence no bookkeeping of the queued updates is necessary. The only needed reference would again be the commit timestamp of the primary node.

While all of these approaches can be used to replicate and refresh the data on outdated nodes, they all come with their own set of trade-offs and might be used in different scenarios. Since all replicas should be refreshed independently, which not only again reduces the total update time but also eases rollback scenarios. However, all are sufficient to fulfill our requirements to even refresh replicas independently from each other *(iii)*. This not only reduces the total update time but also eases rollback scenarios. Otherwise, we might need to define complex countermeasures to undo certain refreshes if one store was already refreshed but another has failed.

4.5 Consistency Constraints

As suggested in Section 2.4, there exist several techniques for how outdated nodes can be lazily updated in the context of freshness. Most of these presented distributed architectures follow a primary-copy approach for master-driven replication to all their secondary replicas. This eases the control and flow of data. Although in the proposed works some systems allowed to access read-only copies directly, with a Polystore, we always have a single point of entry which can vaguely be compared to the polylayer acting as a master node when distributing or even routing queries. However, since any requests have to pass through the polylayer, we have full control on how and where queries need to be routed, allowing us to selectively route read-operations to outdated and up-to-date stores alike. This enables the system to take full control on how different levels of data freshness are being accessed, and which queries are allowed to be executed or not.

Since we have decoupled the update from primary and secondary replicas, we not only need to make sure that they converge towards the same state, but also that all intermediate states conform with each other. This means that refresh operations can only be applied in such a way, that at any given time an outdated version always has to have the exact same state that its primary counterpart had when it was at that time. Without this serialization, it would not be possible to correctly operate and return a comparable freshness-related state. Disregarding the refresh algorithm, we require that all updates are propagated and applied in the exact execution order as they were at the up-to-date replicas. This avoids inconsistencies even when handling outdated data *(iv)*.
Finally, as briefly mentioned in Section 4.4.2, we require a *Refresh-Lock* as a newly introduced locking capability. This lock shall only be applied whenever a refresh operation is currently in place and updates an outdated node. This way the routing mechanism can avoid sending any queries to that replica for reads or a new refresh operation, which might have been triggered manually by an user.

4.6 Transaction Handling

To allow the system to reduce the overall processing time, we need to reconstruct the initial update transaction such that it only targets the replicas labeled as up-to-date. As previously described in Section 4.2, the usage of lazy replication is already sufficient to reduce the strong consistency towards an eventual consistency.
Since Polystores allow us to uniformly access all underlying stores through a centrally defined interface, all requests have to go through this layer and we can easily choose where queries will be routed to. With this abstraction layer on top of the stores, we can leverage the Polystore to act as a coordination service allowing us to restrict modifications to primary nodes only, instead of waiting until an update has been persisted everywhere. Therefore, commonly used update transactions are logically divided into two separate transaction types to allow a deferred refresh of objects. Update transactions in this sense are transactions that contain at least one write-operation.

- **Update transaction**: These contain write-operations that are targeted to primary nodes only and still need to be routed. These originate from a user query in order to modify a data object.
- **Refresh transaction**: These transactions are associated with a refresh operation to replicate pending changes and consequently refresh the data on an arbitrary outdated node. These transactions are normally generated system-internally, and cannot be directly invoked by any user. However, they already have a pre-defined execution plan with a pre-determined set of operations that are going to be executed on outdated replicas.

Although logically being used differently, they are technically executed with the same capabilities and are only really differentiated in terms of their target. Since they do not have technical differences they are rather used as an indication which part of the process is referred to. For data objects that do not contain multiple versions, the update transaction

behavior will not change. With this possibility, we can treat regular queries and queries concerning freshness differently.
Since a Polystore can keep track on which underlying store what part of the data resides, we can redirect all queries to fulfill our intention. Consequently, this allows us to evaluate the freshness and send the queries towards accepted outdated replicas, and further dispatch modifications to designated replicas only.
Since refresh operations are generated based on the original transaction, they already have designated targets and a predefined set of operations. Because this is done prior to the execution, there is no need to route them or identify possible candidates. Consequently, this enables us to employ another transaction aiming solely to refresh the outdated replicas while saving overhead in computation.
Finally, to not interfere with the regular system operation, we require that transactions containing freshness-related queries cannot conflict with the ACID properties of primary nodes. Therefore no write-operations are allowed when specifying a freshness-level, transforming this transaction to a read-only transaction. This is necessary since we do not know during scheduling if the write operation has used results, obtained from an outdated replica. Analogously, when a write-operation has already been executed within a transaction, we can then no longer accept a freshness-aware query. Therefore, the system always has to make sure that the system executes freshness-aware read operations within read-only transactions *(vi)*.

4.7 Freshness-aware Read Access

As mentioned in the previous section, update transactions can now only be executed by users and will always target the primary versions of an object. Hence, they do not allow the specification of any freshness constraints, to ensure the integrity and consistency of the system. Therefore freshness-aware read-operations are restricted to be used within read-only transactions.
Based on the provided freshness metrics considered in Section 4.3 and the different available versions per data object (see Section 4.2), we already have all prerequisites to allow freshness-aware read access. We assume a simple extension of the query language, to allow users to hint or even guide the routing process to identify suitable versions, by defining their tolerated level of outdatedness. Again this could be either done using a timestamp, a time delay, or an artificial freshness-index considering a deviation from the up-to-date replica. On the basis of this specification, the Polystore is able to compare and filter all available versions of the requested object. Although most of the provided related research (see Section 2.7) did restrict the freshness-reads to designated read-only copies, we can leverage the benefits of the Polystore and access all queries uniformly through one single interface. Therefore, if a suitable candidate has been identified within the polylayer, the query can be directly routed toward this replica. Consequently, this abstraction layer also enables us to always fallback to the up-to-date version if no sufficient freshness could be provided among the outdated candidate stores. This efficiently utilizes all sources available to the system *(v)* and omits refreshing an outdated replica, before actually fulfilling the query as described by [53].
For this, we always require that there is at least one up-to-date replica that contains all nec-

essary information, or consequently as many up-to-date replicas that jointly contain all data and no data is lost when accepting outdatedness. This will be verified dynamically when the outdated replicas are labeled, and always enforces these constraints to keep the integrity of the data. Due to the advantage of a central Polystore layer, the routing process can be extended further to support load balancing on the basis of these versions. Given multiple possible candidate replicas for a given freshness selection, the Polystore can monitor and observe if any of these candidates might be currently overloaded and can therefore choose to route the query to a different location. This again harvests the benefits of Polystores and will reduce the latency of such a request.

As with most systems, we might be exposed to different requirements to be fulfilled by freshness-related queries. Although originally introduced to serve especially long-running analytical queries, they might be used in different contexts, hence needing different constraints. One of these requirements is the usage of referential integrity. But despite that a PolyDBMS system might enforce primary-key constraints and hence referential integrity at run time, the usage of multiple versions does not automatically ensure this for outdated versions as well. Although it might be possible to generate dependencies between data objects, such that they need to be refreshed jointly, it should not be generally enforced. Otherwise, it will trigger cascading refresh operations of dependent data objects, neglecting the benefits of decoupling the transactions in the first place. Furthermore, as previously stated, we do not require every data item to exist in several possibly outdated versions. However, if a user wants to specifically use such a constraint even for the outdated nodes, the system will allow this and try to find a suitable combination of all required objects that have been updated jointly. If it cannot identify such a combination, the system can always choose to fallback to the primary nodes successfully serving the query. This is then however done by omitting the advantages of freshness-awareness and employing regular read-operations again. However enforcing referential integrity within the freshness query the system shall be configured to only return equally fresh or newer data, as has already been returned during this transaction. This means that one can only read newer and never data older than you have already obtained. Further, if you need to fallback to the up-to-date version once, all subsequent queries also need to access the primary copy of this object. Although this is not beneficial it will omit the freshness evaluation entirely, hence saving time by avoiding the candidate filtering and pre-selection.

5 Implementation

This Chapter describes an implementation in correspondence to the concepts proposed and elaborated in Chapter 4. These concepts are applied to Polypheny-DB, a particular Polystore system.
First, the current architecture and all relevant components and modules of this system are described. Afterward, each proposition of the concept is adapted so it can be implemented within Polypheny-DB, to enrich it with freshness-aware data management. This chapter is separated into several building blocks, where each part is necessary to describe the implementation in accordance with the requirements. It is abstracted into two main sections. The first addresses the functional requirements *(i, iii, iv)* and aims to apply the concepts of Lazy Replication with all its cross-dependencies, while the second part focuses on introducing the notion of freshness itself, hence aiming to provide the requirements *(ii, v, vi)*. Finally, all building blocks are gathered and put into perspective to describe an entire lifecycle of freshness within Polypheny-DB.

5.1 Polypheny-DB

The implementation is based on the PolyDBMS Polypheny-DB[3], an extended Polystore system. In this chapter, we briefly describe and illustrate a simplified version of Polypheny-DBs current architecture as well as some fundamental components that will be discussed throughout this chapter.
This extends the foundations laid out in Chapter 3 and sets them in the context of the existing system model.
Polypheny-DB is an Open-Source project originally developed as a Polystore system, by the *Database and Information Systems* (DBIS) group of the University of Basel. It has since been transformed into a self-adaptive PolyDBMS that provides cost- and workload-aware access to heterogeneous data [49].
Compared to other systems like *C-Store* [46] or *SAP HANA* [24], Polypheny-DB does not provide its own set of storage engines to support different workload demands.

[3] https://github.com/polypheny/Polypheny-DB

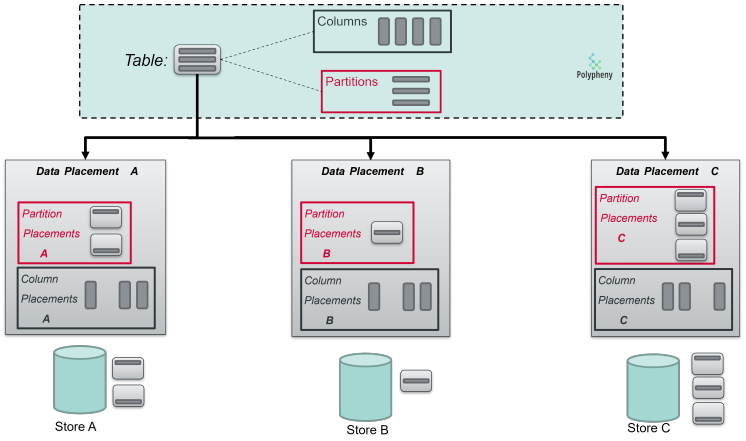


Figure 5.1: Polypheny-DB's Entity Representaiton via Placements. Mapping a Logical Entity to the Actual Physical Stores.

Instead, it acts as a higher-order DBMS, which provides a single-point of entry to a variety of possible databases, like *MongoDB*[4], *HSQLDB*[5], *PostgreSQL*[6] and *MonetDB*[7]. These can be integrated, attached and managed by Polypheny-DB, which will incorporate the underlying heterogenous data storage engines with their different data structures. It is designed, to abstract applications from the physical execution engine, while profiting from performance improvements through cross-engine executions.
For incoming queries, Polypheny-DB's routing engine will automatically analyze the query and decide which store will provide the best response. The query is then explicitly routed to these data stores. This approach can be characterized as a dynamically optimizing data management layer for different workloads. Due to its inherent architecture and the possibility to replicate data across different homogenous as well as heterogeneous stores, it is also able to cluster specific stores on a table entity level, although the underlying stores might not support this natively. This flexibility lifts Polypheny-DB beyond a generic Polystore, towards a data orchestration platform and an actual PolyDBMS [52].

5.1.1 Placements

Placements are considered to be Polypheny's virtual representation of physical entities. Since Polypheny provides a multi model approach, underlying stores can range from relational over document to graph engines. Entities are therefore a general definition encompassing tables, collection, or graphs, respectively of the utilized model. They act as an abstraction between the Polystore layer and the physical representation of an entity. Mostly used within the PolyDBMS itself they help to assist the logical routing process of Polypheny-DB. These different types of placements can be summarized as follows:

Data Placements A *Data Placement* is essentially a virtual representation of the physical

[4] https://www.mongodb.com/
[5] https://hsqldb.org/
[6] https://www.postgresql.org/
[7] https://www.monetdb.org/

entity residing on a given store. A store in Polypheny is an underlying physical data storage which is attached to Polypheny-DB. All attached stores can be used to hold several fragments of data. During routing decisions, stores are automatically taken into consideration if they are designated for the associated data.

A Data Placement contains information on available columns (→ *Column Placements*), partitions (→ *Partition Placements*) as well as properties that are unique to this store. These properties are centrally configured per Data Placement, but will be passed on to the underlying placements as well.

When an entity is created on Polypheny-DB it is an ordinary structure placed onto one store. An entity can therefore contain several Data Placements with different capabilities and properties as depicted in Figure 5.1.

Column Placements Are part of the virtual representation of a physical entity. They are essentially needed to fulfill the intended flexibility of Polypheny-DB. Column Placements are considered to be instances of a column, placed on a specific store. These placements are the result of the extended vertical partitioning of an entity.

As already discussed in 3.1, vertical partitioning refers to the logical separation of the data structure by columns to obtain logically connected objects throughout the database. Polypheny-DB extends this functionality to vertically partition tables column-wise, which allows a table itself to be split further into a disjoint set of columns. Column Placements are instances of a column placed on a specific store. They are considered unique per column on a cluster. Since an entity consists of one to n-columns, in the context of vertical partitioning a subset of these n-columns can now be placed onto another store, which will consequently be part of a Data Placement. This can either be done by evenly distributing the columns onto these stores or by simply replicating the subset to a second store. This functionality enables Polypheny-DB to adapt the data structure to continuously varying use cases.

Partition Placements Although part of the Data Placement as well, a Partition Placement is considered to be the actual virtual representation of a physical entity. It essentially represents and links to the physical entity stored on an underlying engine. Partition Placements are the results of applying a partition function to an entity, to horizontally partition the entity into a distinct set of rows. The resulting Partition Placements can then be placed freely on existing stores to distribute or replicate data. Due to the partition function *NONE*, every entity inside Polypheny-DB is considered to be partitioned, hence consisting only of one partition. Additionally, Polypheny supports the most common partition algorithms like *HASH*, *RANGE*- and *LIST*-Partitioning. It allows to selectively query independent placements and their respectively distinct underlying physical stores, to distribute incoming workload evenly within the cluster. Together with Column Placements, they provide great flexibility to adapt and customize any system to fit various requirements.

5.1.2 Query Routing

The query routing is an essential part of any Polystore system and is crucial for Polypheny-DB's processing capabilities. The routing process can be briefly described as an abstraction layer that will locate and consequently provide the best combination of suitable Data Placements to fulfill a given request or to provide a given result.
The routing process can roughly be distinguished into four phases. A *resolving phase* which identifies individual building blocks, such as partitions, which are necessary for the query execution. The second phase is referred to as *parametrization* and is used to transform the statement into cacheable objects to simplify further routing steps. It is followed by the time-consuming *planning phase*, which generates and proposes possible execution plans to determine on which store and placement combinations a query should be executed. This is finalized by the *selection phase* which is built on top of the previously generated candidate plans and will effectiviely pick the best suitable plan for a given execution. Since especially the time-consuming generation of possible plans can get quite large for highly distributed entities, as they allow for a large set of possible combinations, this phase can be cached. This enables pre-delivering already generated plans for similar queries to be used by the last phase, to save processing time and reduce overhead [48].
Since every query has to go through the abstraction layer to guarantee the correctness and consistency, Polypheny-DB can consult the system internal *Catalog* to retrieve the location of all relevant data. If the requested data indeed happens to be distributed on several stores, the central routing engine will join all relevant and distinct placements to construct the result set. Hence, the query is always routed to stores which hold relevant data.

Given Polypheny's current architecture all incoming queries have to be delivered through this central polylayer, acting as a central coordination instance. Since we assume that there is no direct interaction with the underlying systems, there is no immediate risk of inconsistencies. This allows the utilization of SS2PL to handle concurrency control only within Polypheny-DB for correct isolation treatment. While this provides a serializable execution of each operation by applying suitable locking capabilities, it impacts the availability respectively the latency of the entire system. Since Polypheny-DB has to ensure global consistency, it needs to apply a lock on the entire entity that is accessed. For highly distributed entities, Polypheny-DB has to ensure that every write-operation is applied equally on all stores. This blocks further processing until the operation has been persisted and ultimately committed by all underlying stores. This introduces a bottleneck for this layer, that is inherently dependent on the slowest performing store.

5.2 Lazy Replication

This section discusses all implementations along with the introduced components and services to establish multi-versioning and the possibility to refresh specific replicas. This serves as a foundation in order to use those distinct versions to be used within query retrieval. This again shall help to reduce the overall latency of the system by allowing mixed workload to exist in parallel. In correspondence to the Section 4.4, the update propagtion in our concept

focussed on implementing the CDC approach (→ Section 5.2.6), as well as the on-demand approach using primary snapshot copy (→ Section 5.2.7).

5.2.1 Placement Versioning

As we have established in Section 4.2, the existence of multiple data replicas is fundamental in distributed systems to even provide the possibility of a trade-off between latency and consistency. These versions essentially allow load balancing requests among all suitable replicas to effectively use the entirety of the system. This does not only enable one to distribute the load evenly across the landscape, hence increasing the availability, but also defines how many of these replicas need to be utilized jointly to enforce the desired consistency constraints.
As the name might suggest, a multi-version database would be ideal and the obvious choice for such an approach. These databases will automatically generate a new version per data object for each modification. Due to their properties, we would immediately have the information on the validity-interval of the version, its update time as well as predecessor and successor versions. This would directly allow us to utilize these versions on freshness-related queries. Nonetheless, as stated in Section 3.4, multi-version databases automatically tend to have larger data footprints, due to persisting redundant and even obsolete data. However, Polystore systems already suffer from a larger data volume, given the redundant data storage across several stores. Furthermore, this would also consequently imply the utilization of MVCC. But, as mentioned before, Polypheny-DB currently only supports SS2PL for its concurrency control. Since we require to have equally converging states for our outdated versions *(iv)*, we need a serializable execution that can be applied to the underlying stores as well. Although, MVCC reduces common blocking scenarios and allows write- and read-operations to be executed in parallel, it cannot reliably produce a serializable execution order of all operations among all participating stores. That is why we remain with SS2PL and refrain from using the automatic versioning provided by a multi-version database.
However, as already mentioned in Chapter 4, multiple versions are automatically created when using a lazy update propagation among the participating nodes. This will directly loosen the constraints, imposed on replicas to update. The update in these cases will then only be targeted towards the primary replicas, drastically reducing the response time of a write-operation, but lowering the consistency at the same time. Furthermore, to provide freshness-awareness as well, we do not only require several versions for updates to be applied quicker, but also be able to actually utilize these versions to efficiently operate on the entire system. These versions therefore also allow us to compare and find suitable candidates for freshness-related queries.
Fortunately, Polystore systems and especially Polypheny-DB is inherently distributed, automatically providing potentially multiple replicas. Although they might be distributed or replicated, resulting in redundant data storage, Polypheny-DB allows creating multiple Data Placements for an individual entity. The introduced Data Placements described above can therefore be considered as an individual replica or version of a corresponding data item as referred to in the concept. Therefore, to enable Polypheny-DB to retain different levels of freshness, we need to allow our routing process to only target a subset of all placements

for primary update transactions. The remaining placements will therefore automatically become outdated.
However, since Partition Placements logically refer to the physical entities that actually persist the data, and read-only queries typically benefit directly from data partitioning (see 3.1), Polypheny-DBs Partition Placements are suitable candidates for the implementation of freshness awareness.

5.2.2 Replication Strategy

In order to reduce the update time per write-operation and increase the performance of OLTP workloads, we need to enable Polypheny-DB to identify placements that need to receive updates immediately.
To allow the routing process to differentiate between such placements, we need the possibility to label data placements on how they are going to receive updates. This is defined as the *Replication Strategy* $\Upsilon \in \{EAGER, LAZY\}$. *EAGER* means that modifications are applied at once, while *LAZY* allows data manipulation to be deferred, resulting in outdated data.
Although we defined that we will base the freshness evaluation on Partition Placements, we implemented the strategy per Data Placement. As introduced at the beginning of this chapter, Partition Placements inherit their information from their corresponding Data Placement. Although they receive their updates independently, their properties are defined within the parent placement. Therefore, it is sufficient to configure the Data Placement to achieve an intended state of the subordinate Partition Placements, such that a partition on a given store can either be updated entirely eagerly or lazily.
Since we want to establish the freshness comparison based on each individual Partition Placement, the locking mechanism has been adapted to allow locking on partition level rather than on a table-level. This allows for a much finer level of detail and increases the degree of parallelism.
It can be used to selectively define which Data Placement shall be updated lazily. The replication strategy can therfore be directly defined as:

```
1 ALTER TABLE tableName MODIFY PLACEMENT ON STORE storeName
2                                       WITH REPLICATION ( LAZY | EAGER );
```

Listing 5.1: SQL Statement Syntax to Modify the Designated Replication Strategy for a Data Placement.

This replication strategy is added as part of the newly introduced Data Placement properties. Since Data Placements inherently carry the information, what columns and partitions reside on a given store, they were extended to now also hold information on Data Placement specific properties. When a placement is created without any replication strategy, it will automatically be labeled to receive updates eagerly.
This allows us to flexibly define the strategy per Data Placement, considering all necessary constraints to ensure the consistency and integrity of the data (→ 5.2.8).

5.2.3 Replication State

Since the replication strategy is bound to an individual Data Placement we still need the possibility to define how the actual Partition Placements, that hold the data, will behave in certain scenarios and will consequently be processed. We, therefore, introduce the *Replication State* per Partition Placement.
This replication state is logically bound and directly influenced by the replication strategy defined within a Data Placement and can be differentiated into three states that define the intended state of a given Partition Placement.

UPTODATE This is automatically set within a Partition Placement when the parent placement is configured to receive updates eagerly and cannot be changed by any user. Further, it does not refer to the current state of any data object, meaning that a lazily updated placement can become up-to-date over time. Although this is possible in terms of the received update, it is not respresented using these states. They rather impact the behaviour and handling during processing.

REFRESHABLE Initially, this is configured when the corresponding Data Placement receives updates lazily. This allows the Partition Placement to actively receive individual updates by a replication algorithm. A refreshable state can be automatically and manually transformed into an *INFINITELY OUTDATED* state.

INFINITELY-OUTDATED This state specifically marks placements, to stay outdated and not receive any updates, by suspending all distribution towards those stores. This can either be done manually, because a user may want to retain an item with a given version, hence suppressing the automatic update replication on this node. Additionally, this can be set automatically by the system if either the entire store or the system is not available anymore. This can be caused due to an unexpected outage, or simply because the replication algorithm has numerously failed to apply updates, indicating an error. Given certain prerequisites, it can be manually transformed back into a *REFRESHABLE* state.

The distinction between these cases is necessary, to allow treating Partition Placements on a given store differently. Otherwise, if one Partition Placement would be automatically labeled as *INFINITELY OUTDATED*, the entire Data Placement could not be refreshed anymore. Therefore, they are handled and considered independently.
Although this state is required for internal processing of individual Partition Placements, the manual specification of this state is still targeted to an entire Data Placement (see Listing 5.2). Since the internal partitioning should be rather user agnostic, one should only be able to specify this per Data Placement. As with the replication strategy, the changes are then propagated downwards to all linked partition placements.

```
1    ALTER TABLE tableName MODIFY PLACEMENT ON STORE storeName
2                                          WITH STATE ( REFRESHABLE | OUTDATED );
```

Listing 5.2: SQL Statement Syntax to change the designated Replication State of Data Placement.

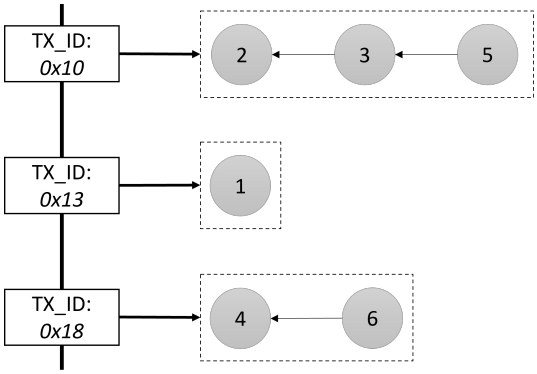


Figure 5.2: Capture Queue, containing Pending Transactions as well as an ordered list of Executed Statements containing the Captured Data.

However, they cannot be set freely and have to follow certain constraints (see 5.2.8). Although placements containing *REFRESHABLE* can be set to *INFINITELY OUTDATED* and vice versa, the state *UPTODATE* can only be influenced by the replication strategy. Trying to change this manually will result in an error since it is controlled by the system.
Furthermore, since each Partition Placement is enriched with the most recent update information to support various freshness metrics, we propose to define the outdatedness on the state of a specific Partition Placement. Although the entire Data Placement could be labeled as outdated or rather receive updates lazily, some of these partitions could already be up-to-date again, while others still remain outdated.

5.2.4 Change Data Capture

Influenced by the replication strategies, the routing process is now capable of differentiating between placements needed to be updated immediately or asynchronously. When processing a write-operation, the router can identify for a given entity if it contains at least one placement that is updated lazily. If this is true, it will capture all executed changes within a *Change Data Object* to be later applied on these placements. Disregarding the operation type $\in \{INSERT, UPDATE, DELETE\}$, it contains information on all logical partitions that have been involved, the executed operation, the current statement as well as the transaction ID. For *DELETE* and *UPDATE* operations it also stores additional information on possible filter conditions. Each statement within a transaction can have at most one of these objects and refers to one operation to be executed.
After creation, this object will be added along with its statement ID to a preliminary *capture-queue* within the *Change Data Collector*. As visualized in Figure 5.2, this capture-queue is represented as a hashtable for faster retrieval, and maps a transaction to a list of statements that require change data capture. These statements are stored with respect to their execution order within the parent transaction. Each statement inside this structure is attached to its respective *Change Data Object*.
To be able to apply operations directly to the outdated replicas, they need to be converted into basic operations that can be applied to a prepared statement. Therefore they are captured before they are executed but after they have been evaluated.

Since not all stores provide the same functional capabilities, we can leverage the Polystore-layer to pre-compute certain calls before applying them to the underlying stores. Typical functions that are not uniformly provided are e.g.: *CURRENT_TIME* or *TIME_NOW*. This allows storing the actual values that are executed on the store, hence saving execution time during update propagation. During runtime of any given statement, the actual evaluated data values are then injected into the object stored within the capture-queue.
The benefit of this structure is that as soon as the transaction commits, the *Change Data Collector* is notified, streaming all objects in correct execution order into the *Replication Engine*, where they will be transformed into individual *Replication Objects* and finally queued to be propagated onto outdated placements. Since the registration is done during the commit, we are sure that any pending changes will be available for distribution once the transaction has been committed.

5.2.5 Lazy Replication Engine

The newly introduced *Replication Engine*, contains the core functionality that transforms capture objects into distinct replication objects and pipes them to specific execution engines. The *Lazy Replication Engine* is a specific implementation of the general replication engine and enhances it with several additional capabilities targeting the lazy replication strategy. This engine essentially provides the CDC approach proposed in Section 4.4.2 and is influenced by the change data capture service of Section 5.2.4, to apply changes operation-wise to designated targets.
During the commit time of a transaction, all corresponding *Change Data Objects* will be first transformed into distinct *Replication Objects*. Other than the generic change objects, these are restructured and specifically tailored to specific operations and designated targets. During transformation, the engine retrieves all relevant placements that are currently defined to receive updates lazily. Then for each of these target placements, an individual replication object is generated, allowing to replicate changes independently from each other. These transformed objects are then added to a *Global Replication Queue* which concludes the change data registration process.
The engine itself is decomposed into the following services, that jointly allow an asynchronous execution of modifications within Polypheny-DB.

Replication Data Object This object contains all information, necessary to re-create a statement which is equivalent to the original one, that has already been executed on the primary placements. Therefore it keeps information on the original transaction, its commit time as well as the operation type and the data to be delivered. This is further enriched with a list of all target Partition Placements that shall receive this modification. The list of targets is generated at the time the initial update transaction has been committed and changes have been queued. In order to avoid storing data redundantly, this data object is centrally stored and only referenced by depending replication objects, disregarding the number of placements that shall receive a given write-operation. This data is kept as long as there are replication objects depending on it for executing their replication.

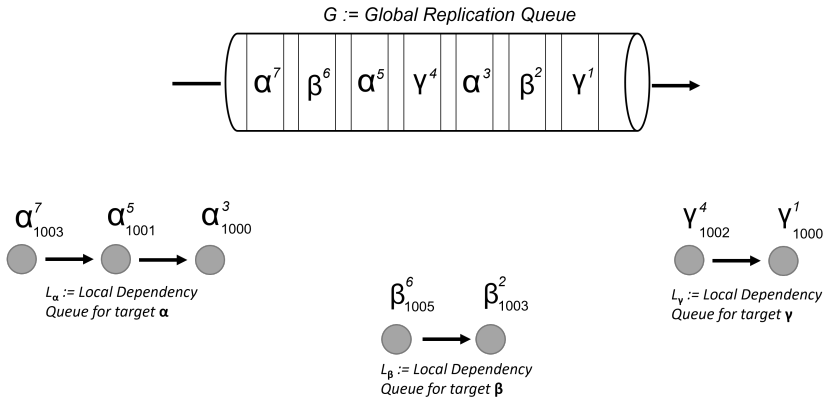


Figure 5.3: Association Between Global Replication Queue and Local Dependency Queues, along with their Replication Data Identification.

Global Replication Queue This queue is the core component and the inherent driver revolving around the lazy replication approach. It contains individual replication IDs, which correspond to replications targeting exactly one Partition Placement at a time. As depicted in Figure 5.3 it is represented as a FIFO queue receiving new replication IDs that are registered through the *Change Data Collector* or are rescheduled by *Replication Workers*.

Each replication event within this queue is therefore defined as x_z^y, where x represents the designated target Partition Placement, y a global replication ID uniquely identifying a specific replication object and z as a reference linking to the actual replication data. Since our replication engine should replicate the data operation-wise, each event within the queue corresponds to one operation associated with one target Partition Placement. Therefore each event can be applied independently.

Since the primary transaction can only be committed when the data to be replicated has been added to this queue, it stores the replication events in-memory to process them faster.

Local Dependency Queue This queue is defined per Partition Placement containing pending updates, that are yet to be replicated. Since all updates need to be delivered in the same order as they have been applied on the primary replica, they depend on each other. Although labeled and represented as a queue, the entries are saved as an DAG, where each replication event depends on its predecessor to be executed first. Since the entries within this queue are not added concurrently and are ordered according to their execution order, we are certain that the imposed dependency reflects the original execution order. If the queue already contains items, new changes will be appended on the left side of the dependency graph. This results in the first new item directly depending on the last item currently in the queue, to be executed first. Therefore this queue is used as an utility to enforce constraints on all intermediate steps ensuring that operations are applied in correct order, that all replicas converge equally towards a given target.

Replication Worker These workers are an essential part of the replication engine since

they continuously process events from the global queue and initiate the execution. As soon as a worker thread has finished processing a replication, it will take the oldest item from the global queue. Before starting the replication process, it will make sure that this is indeed the next replication to be executed based on the dependency constraints within the local dependency queue. If the replication event is not the next to be executed, the worker will re-queue the event, to be processed later. However, if all prerequisites can be verified a dedicated refresh-transaction is started. The one operation is passed on, to the *Data Replicator*, which will then actually execute the statement on a given target. After successful termination, the worker will remove this dependency from the local queue as well as from the replication data.
Based on the current load and the number of pending replications on the system, these workers will be scheduled as needed, allowing to dynamically scale as the system grows. Since the *Local Dependency Queue* always will ensure the correct execution order, concurrent processing of the replications is possible.

Data Replicator Implemented as the actual execution engine, the replicator will recreate and execute a captured operation. It is invoked by the replication workers, passing a replication event together with a reference to the corresponding replication data. For the Lazy Replication Engine, it will receive one operation per target placement. This has the advantage that we can now decide, based on the target's current placement structure, how to reconstruct the statement. Because this is done right before execution, it shows the benefits of capturing the entire object instead of the statement to be executed per placement. Since this target placement could have been structurally altered since the initial capturing, it could now have a different set of columns. This allows the replicator to adjust the target statement on time.

Update Metadata Since data is essentially stored on physical entities, represented by the internal Partition Placements, we extended these objects to contain information on general update statistics relevant for this particular placement. This information will be updated at the commit time of a write-operation. For eagerly replicated placements this is the primary transaction, for lazily replicated however the refresh transaction. This enables the system to use this update information to essentially retrieve the current state of the data, represented by the commit timestamp and the number of updates this Partition Placement has already applied. Allowing us to use those metadata to compare different versions against each other.

With all these services the replication engine is able to process multiple replications at once while ensuring the correct execution order and stability for each replication to be executed. Since all operations are propagated atomically within one transaction we do not need to worry about a rollback of a refresh transaction or provide complex undo-operations to remove those changes. Therefore, these replications can be applied independently per target Partition Placements and do not interfere with each other, allowing for high flexibility.
Furthermore, it does not only replicate the captured modifications it provides a fault-tolerant approach as well. Since Polypheny-DB consists of multiple attached stores that might suffer from outages or local failures, replications might fail. However since the replication data is

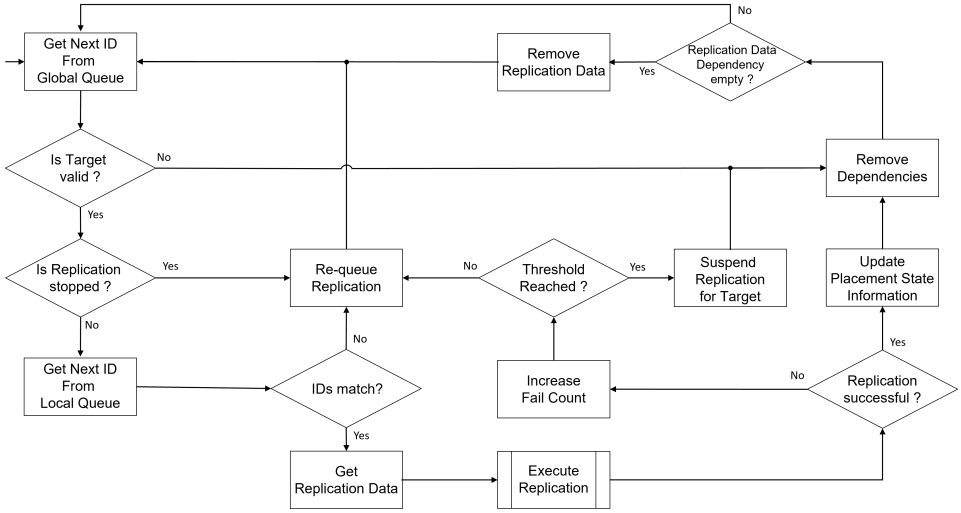


Figure 5.4: Lifecycle of a *Replication Worker*, Processing Pending Replications from Central Queues.

centrally stored and is only removed when all dependent replications have been executed, a replication event that continuously keeps failing will unnecessarily occupy worker threads and waste storage resources to preserve the data. Therefore a centrally configured *Fail Count Threshold* is introduced. Every time, a replication fails, the responsible worker thread will increase the *Fail Counter* of a given replication object. If it reaches the threshold the replication will be removed from the queue entirely. Furthermore, the target will be removed from the engine by deleting all remaining local replications for this target as well. Consequently, this particular Partition Placement will be suspended for active data capturing and marked as *INFINITELY OUTDATED*.

Due to the architecture of the queue, we can only get an event at the top of the queue. This increases the difficulty to remove all associated replications for a target placement at once. However since workers will validate if replications can even be executed and can therefore identify suspended targets, they will automatically cleanse the queue from unwanted replications, without expensively removing each item within the queue.

Moreover, contentions can also occur directly within Polypheny-DB, if there are too many pending updates, waiting to be replicated. Despite the scaling capabilities of the workers the load on the system might still be too high, essentially affecting the main operation on the system. Therefore we have enriched the configuration to globally enable or disable the replication distribution. In contrast to labeling Data Placements as outdated, and removing all pending replications forever, we can now allow to temporarily stop the replication and only continue to capture modifications. This can help the system to adapt to high-load situations by focusing on its core functionality.

5.2.6 Automatic Lazy Replication Algorithm

The goal of the entire lazy replication approach is to provide a scalable and fault-tolerant approach to distribute the data for each placement onto the designated stores. Therefore,

the algorithm as depicted in Figure 5.4, aims to provide a cost-efficient approach to replicate the change-data without increasing the overhead of the system and impacting regular operations. For every write-operation, the routing process will verify which placements need to receive this modification. During this process, all eagerly replicated stores for this entity are identified. Since all entities contain a list of all Data Placements, we can compare the delta between the retrieved stores and the actual stores. If there is indeed a delta, we can conclude that there are placements which consequently have to be updated lazily. That enables the entire transaction to *capture change-data*. This will directly result in transforming the write-operation of the current statement into a set of basic operations, that are already evaluated and can therefore be immediately applied to target placements. Consequently, a *Change Data Capture Object* will be created, containing all information needed to recreate the statement again. Accompanied with the ID of the current statement, as well as the parent transaction, this object is prepared and appended to the capture-queue.
During runtime, when the statement is about to be pushed-down to the designated stores, the prepared statement is enriched with the pre-evaluated information, necessary to execute the statement. These parameter values are added to the *Change Data Capture Object* as well. Accompanied by the parent transaction and the statement ID, we can identify this change object within the preliminary capture-queue and enrich it with the necessary information (see Figure 5.2).
As soon as the transaction has finished, this object is further processed. If the transaction was aborted and rolled back, we again can directly remove all pre-queued changes that are associated with this transaction by removing the entry from the nested hashtable. This will cascadingly remove all attached capture objects.
However, if the transaction will be successfully committed, the finalization phase starts. For all capture objects associated with this transaction, the corresponding commit timestamp of the transaction will be set. This can later on be used for freshness comparison. Afterward, each capture object will be registered at the *Lazy Replication Engine*. To do this, the joint capture object will now be separated into two parts. The first one is the replication data, which contains all information to be applied to secondaries as well as all Partition Placements that shall receive this replication and thus depend on this data. The second one is the creation of independent replication objects targeted to individual Partition Placements. They are bound to a specific operation and responsible for a given target. Additionally, they are linked to the single replication data for this change-data set. This allows us to reduce the data footprint by caching the replication data only once. The list of individual replication events as well as the data is now passed on to the queue registration. This consequently adds for each target placement a new entry into the global queue. Additionally, it also appends the corresponding replication ID to the local queue of each Partition Placement. Once all replication objects have been added to the global queue the finalization phase ends.
Since these capture objects have been added to the preliminary capture-queue in their execution order, they are also passed on to the actual queue in this exact same order. This ensures the consistency among the different placements and enforces that they uniformly progress towards a given state as their up-to-date counterpart has.
Since this is all still done before the transaction has publicly finished, further operations

are blocked until we have assured that all objects have been consequently transformed and queued within the associated replication engine. Thus we can ensure the consistency of the secondary placements by waiting until all steps have finished successfully. If something would have gone wrong during queueing, we can still relabel those placements as *INFINITELY OUTDATED*, marking them as not receiving any more updates, hence informing users and administrators that this placement will stay outdated until manually fixed.
After the queueing process has finished and the primary update transaction has returned, removing all locks, the *Replication Workers* will eventually process the queued events as illustrated in Figure 5.4. As soon as a worker has free resources again, it will take the next item out of the global replication queue and starts analyzing it. For each item, it will verify if this is indeed the next replication to be processed for this target, by obtaining the next item of this Partition Placements local queue. Additionally, it verifies if the replication distribution has been suspended entirely. This will lead the worker to reschedule the replication and append it at the end of the global queue again, and move on to the next item.
However, if this replication does not exist anymore or this target has been marked as *INFINITELY OUTDATED*, avoiding additional replications to be applied, the worker identifies it as invalid and automatically cleanses the queue from all remaining replications associated with this placement.
Assuming that the currently processed event is indeed the next replication in line it will prepare the execution and start a new refresh transaction. It is handed over to the *Data Replicator*, which will now analyze the replication event, and ultimately reconstruct a new modification statement that will be routed internally towards the designated Partition Placement.
When the replication has finished, the item is removed from the local queue. Additionally, it removes itself from the dependency graph of the associated replication data. If the corresponding replication data does not have any dependencies left, it can also be safely removed to reduce the data footprint of the system and free up resources again. However, if the replication process for this target fails, the centrally defined fail count for this specific replication event is increased. If it is above the configured threshold, the system will abort further replications of this target placement, labeling it as *INFINITELY OUTDATED* and removing all pending replications of this replica. Otherwise, the replication has been successfully executed.
Because the presented lazy update propagation is done operation-wise, we actually loosened the heavy SS2PL constraints that we require on the primary updates. Since we already have a serializable schedule after execution, we also know per entity and each Partition Placement the correct execution order that we have to apply the data changes on. So it is not necessary to only free the resources after the entire transaction has been replicated but right after each operation. However, since Polypheny-DB currently only supports a SS2PL approach, we can mimic this behavior by scheduling refresh transactions containing only one modification. This allows us to replicate data using the less strict 2PL approach, hence improving the overall performance of replications with respect to the primary execution.

5.2.7 Manual Refresh Operations

Although, the automatic refresh operation, based on the change data capture approach will continuously propagate incoming changes, it is not able to prioritize certain placements after they have been added to the queue proposed in 5.2.5. Since users might need to be able to specifically prioritize Data Placements or entire tables to be updated faster than others. This could also be the case if one placement is currently marked as *INFNITELY OUTDATED*, either because of too many failed replications or because it was manually configured to remain in a given state. As described in 4.4.2, we need to be able to provide manual refreshes on the basis of *Primary Snapshot Copies*. This inherently uses the capabilities of Polyphenys existing *Data Migrator*, which essentially queries a defined source entity on a given store. Together with the current *Update Metadata* of this placement, the extracted result of this query can be considered as a snapshot of a given replica. The contents of this result will be subsequently applied to a targeted placement.

After the data migration process has finished, the target receives its new update information on the basis of previously extracted metadata. If this placement had remaining replications in its local dependency queue, they are all removed to avoid adding data twice. The system then automatically switches the replication state of this placement to *REFRESHABLE*, starting to actively capture changes again. Since the snapshot merely depends on the source, only a short-lived shared lock is applied until the data has been extracted. Apart from this, the snapshot will have no impact on the actual system.

Despite that data resides on the partition placements, users may only directly interact with a Data Placement of an entity. We therefore either allow to manually refresh one specific Data Placement respectively all Partition Placements residing on a given store or an entire table.

```
1 ALTER TABLE tableName REFRESH ( ALL PLACEMENTS
2                               | PLACEMENT ON STORE outdated_store );
```

Listing 5.3: SQL Statement Syntax for an On-Demand Refresh Operation

Although such refresh-transactions can be executed on any placement, they will have no effect on primaries. The same holds for placements that are already up-to-date with respect to their corresponding primary-version, where the execution will be simply omitted.

5.2.8 Placement Constraints

As Polypheny-DB is designed to be a distributed system, we always need to ensure that we do not lose any data, while transforming the individual placements. This already starts by defining the replication strategy. Although Polypheny-DB allows to customly distribute an entity across several stores, we have to ensure that no information is lost. This means that since we can arbitrarily place any combination of columns and partitions of a given entity of any store, we need to make sure that in the end, each column is represented by all available partitions at least once. Otherwise, this would violate the integrity of our system. This consequently needs to be considered for outdatedness as well. Since we have decoupled updates of eagerly and lazily replicated placements, we again could lose data. Therefore we have to ensure that at least the eagerly replicated placements are sufficiently configured

such that each column is available for all partitions at least once. The remaining secondary placements however can again be arbitrarily combined without any requirements.
Because it is possible to switch freely between *LAZY* and *EAGER* strategies even after they already contain data, we again have to verify that no data is lost. Therefore, when trying to switch from *LAZY* to *EAGER*, we have to ensure that this placement does not contain any pending updates, otherwise the operation will fail. If there are currently no pending updates the system will lock the entire table, so it will not receive any updates while switching the strategy internally. Since this is merely done by setting a flag, the impact of the blocking behavior can be neglected.
Furthermore, as stated in 5.2.3 it is possible to switch the replication states of all Partition Placements of a given Data Placement from *REFRESHABLE* to *INFINITELY OUTDATED* and vice versa. While the manual switch to *INFINITELY OUTDATED* is an intentional suspension of the replication procedure, the other direction requires validating possible deviations. If this is not correctly ensured, and the replication starts propagating changes towards this placement again, we will lose the data in between those versions. In this scenario, the system first will need to make sure that the placements on this store are all up-to-date. If this is not the case the operation will fail. This can also be done manually by executing a *Primary Snapshot Copy* as described in 5.2.7. A subsequent change of states will then be accepted.
Finally, despite its ability to remain operational during failure situations, the utilized queues within the engine are still only stored within main memory. Although this decreases the overall commit time of a transaction it results in a loss of all captured data that has not been applied yet, as soon as the system shuts down for any reason. Since we rely on the fast registration process to commit the primary transaction as fast as possible we need to mitigate this behavior. Since we always know which placements are updated lazily, the replication engine can immediately provide two different mechanisms that can be centrally configured. The first approach is simple and fast to execute. It gathers information on all placements that need to be updated lazily. It can automatically mark them as *INFINITELY OUTDATED*, and therefore suspend any more replications. Users can then only choose to update the placements manually. The second approach however results in a much slower startup time by refreshing all placements first, before the system will even start.

5.3 Freshness Awareness

With the Lazy Replication and the accompanied deferral of transactions prepared, we have already established a foundation to use those possibly outdated versions for freshness-awareness. This section, therefore, focuses on all aspects to establish the core aspects of freshness within Polypheny-DB, to allow users the specification of freshness and efficiently using those outdated placements to distribute the workload across the system. The actual freshness evaluation can therefore be separated roughly into two phases. While the first one uses the specified freshness tolerance to generate an applicable filter to identify suitable candidates, whereas the second phase is concerned with constructing and selecting possible combinations of outdated versions to provide an efficient workflow.

5.3.1 Evaluation Types

As delivered through the concept in Section 4.3, there exist several possible freshness metrics to use for version comparison. These metrics are primarily used to filter placement candidates based on a defined freshness using their *Update Metadata*. We have summarized these possibilities and established three different evaluation types, that can be used to specify an accetable level of freshness.

Timestamp A timestamp is the most simple form of a freshness evaluation type since it intuitively provides the capability to define an acceptable lower bound of outdatedness, for a specific query. Potential Partition Placements are therefore filtered by verifying that their commit timestamp is newer than the specified one. Otherwise, they are removed from the list of possible candidates.

Time Delay The time delay can be specified together with a time and an associated time unit to define an acceptable delay for the desired freshness. This will intuitively subtract the specified *Absolute Time Delay* from the current time generating a lower bound timestamp. This again allows filtering all Partition Placements based on the age of their commit timestamp. As described in Section 4.3, an absolute time delay based on the current time is not always accurate or might even render wrong results. Therefore, a *Relative Time Delay* specification is provided as well, allowing to specify the tolerated time deviation based on the commit of a primary placement compared to an outdated placement. To differentiate those options, they are individually suffixed with either *ABSOLUTE* or *RELATIVE*.

Freshness Index An index naturally expresses the freshness specification based on the modification deviation between an up-to-date placement and a refreshable placement. The *Modification Deviation* therefore allows comparing the number of modifications of specific Partition Placements to define a freshness index. For a query, this index can be either configured to filter based on each partitions placements deviation or on their accumulated deviation. Therefore, the number of modifications is accumulated and then compared against the accumulated number of modifications of all up-to-date entities that have been considered within this query. This allows a more stable approach since it is not prone against outliers. Furthermore, this index can also be configured using the freshness index to evaluate the *Time Deviation* between two replicas. Essentially the index dictates how accurate a given placement is with respect to its up-to-date replica.

Since these evaluation types are fundamentally different in terms of version comparison, they allow users to indicate their tolerated level of freshness, depending on their requirements or preference.

5.3.2 Query Specification

Equipped with the evaluation types, users now need to be able to define their intended level of freshness. Although this could be centrally defined for an entire system, it is very

subjective and often does not even depend on the user but is rather influenced by application-specific requirements. Therefore, the specification should be rather defined on a query level, to allow a more fine-grained definition.
This can be achieved by extending the query functionality of Polypheny-DB, allowing users to directly specify their demands. Although Polypheny-DB provides multiple query interfaces and languages, the following specifications are solely demonstrated with SQL. As proposed within Section 5.3.1, we have introduced three freshness types that can be used to specify a tolerated level of freshness. These support all freshness metrics as described in the corresponding Section 4.3. Along with the functional requirement *(ii)*, all these metrics have been introduced within Polypheny-DB.
For SQL, the query syntax is extended with an optional leaf expression at the end of every query to generally support freshness specifications (see A). Along the description, users can choose to select any of the presented evaluation types to guide the system if and how it should consider the freshness of an entity.

```
SELECT * FROM tableName [ WITH FRESHNESS [ <TIMESTAMP> | <DELAY> | <INDEX> ] ];
```

Listing 5.4: Generalized Freshness Specification

The statement depicted in Listing 5.4 illustrates a generalized and abstracted freshness specification including placeholders for the different evaluation types.
While a statement with an explicit evaluation type provides a direct intent, users can also choose to solely specify *WITH FRESHNESS* and omit the type, to implicitly suggest that you will consider all outdated data regardless of the actual version.

5.3.3 Freshness Processing

As for every query, a freshness-statement is first transformed into a tree, parsed, and evaluated in terms of syntactical correctness. Additionally, the query is analyzed semantically where also the specified freshness will be extracted. The designated *Freshness Extractor* validates, if the provided freshness evaluation type even exists and if the specified values can even be associated with this type. Furthermore, it verifies if the specified level is in bound of the possible value spectrum. Otherwise, the statement cannot be executed and will be canceled.
If the query and the freshness specification have been successfully analyzed the extractor generates a designated *Freshness Object*, containing all specified freshness details. This will be attached to the statement to be used for further processing. It automatically helps to enrich the statement with the correct freshness information and informs the transaction that freshness is being used, which directly influences locking capabilities and changes the routing process. Along with requirement *(vi)* as stated in Section 4.6, this will transform the transaction to read-only mode, prohibiting the execution of any more modification statements. This is necessary to avoid that a write-operation uses outdated data retrieved within this transaction to update a placement.
Since freshness-awareness allows to read data that is considered to be stale, we implicitly support *Dirty Reads*, hence violating the ACID properties. Therefore the read-only

transaction now allows to omit locking entirely, by enabling the new isolation mode *None*. Compared to the conventional isolation mode *Serializable*, which enforces locks on the basis of SS2PL to provide correct concurrency control, *None* loosens all constraints. This causes refresh operations to override or add new results to the entity while it is currently being read. Besides the obvious benefits of allowing more parallel load on the system, it has a positive effect on the replication as well. Since the provided CDC approach is designed to continuously provide the placements with captured updates, the target placements will be almost consistently locked due to the number of write-operations. With a *Serializable* isolation level, those replicas could not be used for freshness-related queries after all. This would completely mitigate the performance gained by decoupling the transactions in the first place. Therefore, if not centrally configured or explicitly specified otherwise, every freshness query will natively omit the lock acquisition to provide even more parallel workload processing.
Although, not necessary for all queries, users still might desire to read outdated data without having to fear that the data will be refreshed while reading. Since Polypheny-DB essentially uses SS2PL, shared locks can be easily extended with more transactions adding themself to the list of already waiting transactions. Despite that this avoids *Dirty Reads* and the results will stay stable and consistent, it will block further refresh operations on this placement entirely.
This however could then lead to starvation of the refresh transactions. Therefore we embedded an extension to this locking approach, especially for this use case, which does not interfere with the locking mechanism that targets primary copies. When there is currently a shared lock on an object that is about to receive a manual refresh operation, this disables the possibility for any new upcoming freshness transaction to add itself to the existing shared lock. Instead, they are treated and added to the dependency graph of the refresh operation and its Refresh-Lock, as if there would already be an exclusive lock in place. This would avoid starving the refresh operation while still being able to serve freshness queries on different placements. This can be further enhanced by centrally configuring a threshold, which defines after how many retries the refresh operation can finally force itself to be executed.
While the previous approaches simply tried to provide any possible combination and construct a result out of different independent versions, the last characteristic goes a step further and allows to specify referential integrity enforcement. As described in Section 4.7, this approach aims to enforce the referential integrity among all entities used within the transaction, allowing to retrieve a cross-entity result that was valid in the past. Since it builds on top of the previous characteristic it also needs to lock the placements. However, in our implemented scenario, other than in classical multi-version databases, we are not even guaranteed that every entity has several versions. Therefore, we require for such a query that all relevant placements need to be updated by the same original transaction in order to guarantee that this state is valid. However, if there is doubt or no suitable combination of placements can fulfill this request, we always have the possibility to fallback to the primary nodes, which are guaranteed to support referential integrity. Although, this will again block updates on the primary placements, it fulfills the constraints as it would have when executing the same query without specifying a tolerated level of freshness.

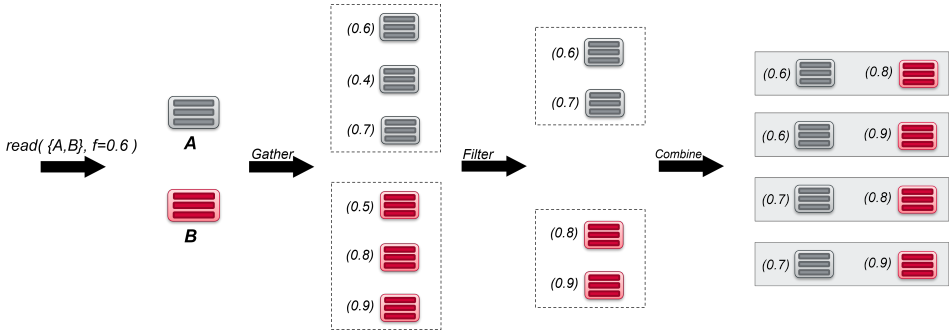


Figure 5.5: Subsequent Filter Operations for an abstracted freshness read operation, generating several possible execution plans based on a given freshness-index.

5.3.4 Freshness Filtering

The *Freshness Manager* is the heart of the freshness evaluation and will consequently assist the routing and selection process to propose eligible placements to be used for retrieval. It aims to apply and analyze the freshness specification captured within the *Freshness Object* of a given statement and transforms it into a general filter condition (*Freshness Filtering*). These filters will be directly derived from one of the selected evaluation types described in 5.3.1. Since this is done centrally during processing, it can be applied arbitrarily, disregarding the underlying stores.

Initially the routing process will first assess which partitions of the requested entities are needed to provide a result. This is an independent step and will be done regardless, whether a regular or a freshness query is processed. A list of all required partitions is then passed on to the *Freshness Manager*, along with the associated Freshness Object. As already described above, we have applied our versioning on the basis of Partition Placements, since they represent the actual physical tables, the *Freshness Filtering* and evaluation are therefore always executed by comparing the *Update Metadata* of individual Partition Placements. Disregarding the actual type, the Manager then retrieves all Partition Placements for each required partition that are considered to be updated lazily. As previously stated, this can be easily retrieved since these are represented by the Replication Strategy: *LAZY*, which is configured within each placement. This operation will consequently generate a data-structure, mapping the required partitions to a list of potential Partition Placement candidates.

With these candidates, the type-specific filter function is invoked. Each evaluation type and freshness specification has its own filter functionality and processes the placements differently. This is mainly done using metadata within the update information properties of each Partition Placement. These already contain information on the current commit and update timestamp, the original parent transaction which has updated this placement as well as the number of modifications this particular placement has received. This information can be consequently used to compare their state against the corresponding primary placement holding the same partition. While *TIMESTAMP* and *DELAY* can be used directly to assess the candidates, the utilization of an *INDEX* needs an intermediate computing step. Disregarding the delay being based on modification- or time-deviation, we have to calculate, per potential placement, its designated index. Naively, the filtering can then be done by ensuring that placements are only kept among the candidates if they fullfill the constraint

and are above a given tolerated threshold. However, for the deviation types, we also allow to assess the freshness jointly among all possible candidate combinations. This means instead of filtering on the basis of single candidates we extend this notion to be able to filter based on their accumulated freshness when combining the placements (see 5.3.5). Since our query can require multiple partitions at once, we can now combine a very fresh placement with a placement that is by itself considered to be too outdated. While the individual placements might not fulfill the constraints, their combination might do, allowing to consider more candidates, hence tolerating outliers. Due to the possible strong fragementation of versions with this approach, it is not used by default and needs to be configured separately. Based on the final filter application or already on the initial pre-mapping of partitions we can respectively identify if we have sufficiently collected Partition Placements for all partitions. If one map should be empty, the freshness manager will halt and automatically fallback to a regular routing approach, targeting only primary placements as if freshness has never been specified. Since theoretically an empty partition mapping would not require the entire processing to fallback to the primary version, and would be able to solely provide a primary copy for this partition alone and still allow to construct the freshness, for the remaining partitions. However, we need the system to provide a seamless and reproducible approach for users, and strictly distinguish between regular and freshness operations. Additionally, as defined with requirement *(vi)*, we should not interfere with the primary versions of the system, resulting in a defined fallback scenario whenever the freshness criteria cannot be uniquely fulfilled.

5.3.5 Freshness Selection

The combination of possible candidates is also executed within the *Freshness Manager* and builds on top of the candidates proposed by the *Freshness Filter*. It tries to combine accepted candidates with each other, in order to provide possible executable plans, as visualized in Figure 5.5. While this step is trivial if we only have one possible candidate per partition, it becomes more complicated for entities containing multiple placements with several partitions. This could easily result in large permutations and possibilities to combine the different placements with each other. Therefore, selecting a suitable combination is often time-consuming and can lead to performance impacts, resulting in freshness queries to be less efficient than regular queries due to the enlarged processing times. But, since the introduction of a freshness notion is inherently driven by the pursuit to reduce locking and speed up queries we have to retain additional overhead as much as possible. While the first steps during filtering need to be applied for every freshness query, the combination step should therefore be executed as little as possible.
As discussed in Polypheny-DBs architecture (5.1.2), the routing is essentially decoupled into four parts, whereas the *planning part* is the most complicated one, since it also needs to generate potential plans which can later be used during the selection phase to be executed. Because this is very time-consuming, it allows utilizing pre-cached plans to skip this expensive phase. Therefore, our freshness combination process should be attached to this phase as well.

If based on the filtered placements, the *Freshness Manager* now recognizes that there are too many possible combinations and the planning would exceed a common processing time, the actual planning and combination phase is deferred to be executed asynchronously. However, a suitable substitute plan needs to be provided to still fulfill the request. While it is always safe to simply return the result generated by a primary execution plan, we can also choose to simply select one of the generated candidate plans, before the *Freshness Manager* has decided that this computation is too expensive and will be deferred. The deferred planning phase can then be executed at a later point in time, to decouple the time-consuming combination of candidate placements from the actual execution. These generated plans associated with a given query can then still be added to the cache, to be reused next time. If during processing we then indeed can involve cached plans for a similar query, the planning can be omitted.
The selection of these cached plans and their encompassed placement combinations then ultimately depends on individually specified attributes. These could be allowing dirty reads, ergo allowing refresh operations to be executed during active reads on outdated nodes or if we rather need a soft locking approach here as well. This would forbid reading from placements that are currently being updated or forbid updating placements that are currently being read. However, this again reduces concurrent writes and reads on the store, imposing possible downtimes and reducing the availability trait.
Since the *Freshness Manager* always identifies the given freshness first and gathers all possible placement distribution possibilities for a given freshness specification, we can validate if we have already cached a similar query where we had the same placement distribution as input. This allows us to use the cache, skip the planning phase and avoid generating possible combinations again, to provide access to outdated data.
This ultimately enables freshness-awareness within Polypheny-DB, to evaluate and utilize the freshness specification, to improve the inherently distributed architecture, to decouple transactions and efficiently distribute workload to also utilize outdated replicas and enhance the entire processing capability of the system.

6 Evaluation

This chapter is separated into two sections. The first section aims to validate and verify the implementation in terms of its correct execution. It focuses on the core contributions, such as the lazy replication algorithm as well as the freshness filter capability. Further, it ensures the correct handling of the described constraints and establishes certain test cases to identify possible failures that can occur during execution as well as suitable failure handling scenarios.
The second part of this chapter focuses on benchmarking the implementation based on the individual building blocks themselves. This is followed by comparing the performance of several scenarios to identify how the system will behave in certain situations. Further, it will compare these executions to determine how the implementation fulfills the provided requirements. Finally, the performance of the individual functionalities is benchmarked and compared jointly to give a comprehensive overview on possible performance impacts.

6.1 Goal

The evaluation has three goals: The first goal is to verify the correctness of the solution, the second goal is identifying the impact freshness-aware aspects have on different workloads and finally the identification of suitable workloads for freshness-related queries.
We therefore want to verify and validate the correctness as well as the completeness of the implementation based on several characteristics. These include the correct execution of the automatic lazy replication procedure, the possibility to refresh statements on demand as well as correct freshness filtering.
Since one of the main goals was to relax the consistency to allow parallel workloads, we want to identify possible impacts of the replication engine on the overall performance if freshness indeed increases the possibility for parallel workload on the system. Finally, we want to observe if the solution allows harvesting benefits of the underlying stores to adapt to various situations to assess the benefits of the Polystore in general. We therefore, focus on evaluating the essential functionalities to provide a foundation for future assessments.

6.2 Correctness

The correctness of the introduced solution mainly focuses on two parts. For one the replication behavior, to verify if each replication is carried out correctly and if not verify that reasonable countermeasures are available and apply them. This is crucial since we do not compare the footprint or the integrity of the data after a replication update. Rather we compare metadata of two replicas on a high level, i.e. if the number of modifications and the commit timestamp after the data replication are equal on the primary and secondary node.

The second part of the validation process focuses on the retrieval of outdated nodes. Although we always have the possibility to fallback to the primary placements as described in Section 5.3.5, we still want to avoid excessive locking to parallelize requests to ultimately speed up the average response time.

As mentioned within the implementation chapter, several constraints will already be enforced with the execution itself (see 5.2.8). We have e.g. established a dependency between the replication strategy as well as the designated placement state. When altering the strategy of an object we automatically ensure that the state is altered as well. This does not only ensure that the placements will be addressed correctly, but also ensures that no data is lost. For example, it enforces that a lazy replicated placement, that still has pending replications, cannot be switched to now receive updates eagerly.

Furthermore, since we have to make sure that no data is lost in general, we have established a strong dependency queue that will apply the update operation-wise in terms of the execution order. Since we cannot avoid that stores might fail, we have ensured that failing updates will not block the replication queue or will be applied in atheorder. Accompanied with a failure-threshold and the placement role *INIFINITELY OUTDATED*, we can instruct the system to mark this placement as permanently outdated avoiding further updates. This ensures that all statements labeled as *INIFINITELY OUTDATED* will always be treated correctly.

The freshness specification in general ensures that only acceptable values are considered for the respective evaluation type to propose the correct results and translate them into a given tolerance value. Since freshness operations violate the ACID guarantees and we will read stale data, we ensured that these can only be executed within read-only operations, such that they do not override data.

Additionally, to guarantee the correctness of the implementation, several unit-tests have been created to ensure the correct handling of data, when using freshness-awareness or replication approaches in general. These logically include a purposely false execution of the described constraints to ensure that they will be treated correctly.

6.3 Benchmarks

The second part of this chapter focuses on the performance impacts of the introduced solutions. The following steps outline the procedure for benchmarking data freshness within Polypheny-DB.

Each evaluation step will subsequently address each required building block to provide data

freshness. Each building block is checked separately, and then put into perspective of the entire context. These benchmarks progressively build on top of each other and are finally summarized with a compound benchmark.

6.3.1 Evaluation Environment

For all executed tests a base set of functionalities has been used to ensure the reproducibility of the benchmarks.
If not explicitly stated otherwise, the benchmarks will be executed using two underlying stores. For these stores, HSQLDB and PostgreSQL are used. While HSQLDB will run natively on the system, PostgreSQL is configured within a virtualized container environment with limited resources imitating a weaker performing store. Although not explicitly mentioned, each test will always be pre-configured to match the required replication strategy.
During the evaluation, only a predefined set of workloads will be used to obtain insights on various setups. As suggested in Table 6.1 these inherently differ in terms of their focus on write- or read-operations as well the degree of utilized freshness queries.

Workload	**Read %**	**Write %**	**Freshness %**
Write Only	0	100	0
Mixed Workload	60	40	0
Read Only	100	0	0
Freshness Only	100	0	100
Mixed Freshness	60	40	100
Partial Freshness	100	0	50

Table 6.1: Available Benchmark Workloads

To further assist the execution of the benchmark tests, a number of tools are used to standardize and ease the evaluation procedure. These tools are described in the following two sections.

6.3.1.1 Chronos

Chronos[8] is a toolkit which enables users to define, monitor and analyze the results of an evaluation for several database systems [51]. It encompasses several tests and evaluation steps, which will be accumulated in terms of the runtime per target. This runtime information can then be used to compare and visualize the differences between different execution environments.

[8] https://chronos-eaas.org/

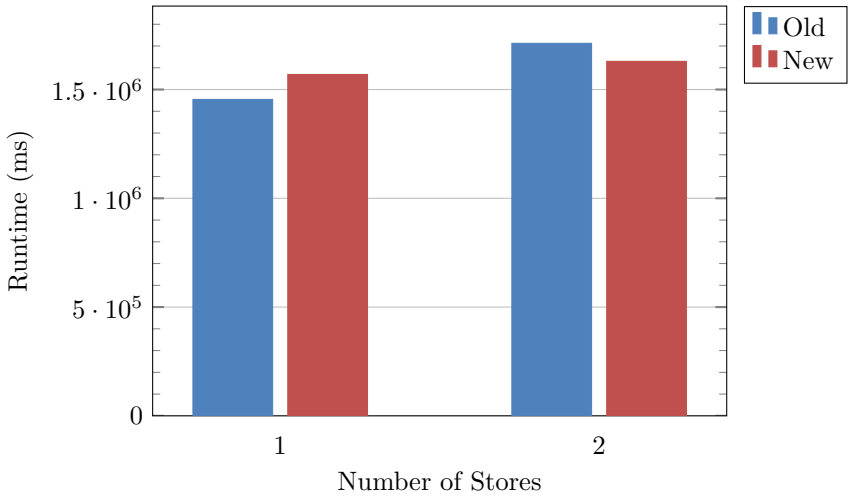


Figure 6.1: Overhead Comparison of Old vs. New Implementation.

6.3.1.2 OLTPBench

For all considered functional tests *OLTPBench*[9] has been used. This benchmarking framework bundles several workload classes and encompasses multiple benchmarking tools to seamlessly provide benchmarking for various DBMS [21].
To get a more in-depth view of how the implementation affected the overall performance, all tests have been executed using *YCSB*[10]. *YCSB* not only offers a fundamental correctness test but additionally simulates workload on a single table to obtain essential performance baselines [17]. These can be used to retrieve performance metrics and obtain indications regarding correct statement execution and query handling.

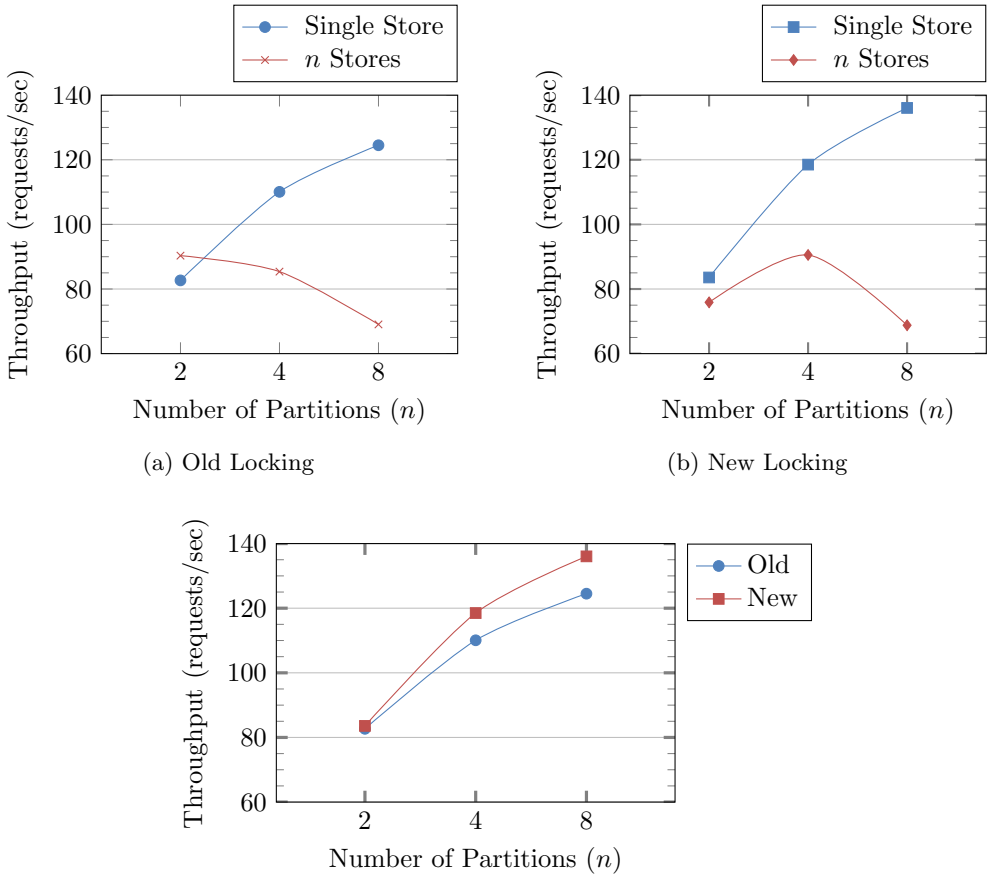
6.3.2 Results

In this section, we will present the results of the performance evaluation. If not stated otherwise the generated tables within the benchmarks will contain 100000 entries and will be executed using 50 parallel client threads on YCSB together with the workloads defined in Table 6.1. During the evaluation, it will only be referred to the respective workload names.

6.3.2.1 Overhead

Polystore systems and specifically Polypheny-DB uniformly collect all incoming requests, process them and, then route the resulting queries to the designated stores, where they will be finally executed. However, due to this centralized processing, additional introduced overhead within this layer will directly impact the performance of the system. Although, Polypheny-DB aims to provide cost- and workload-aware self-adaptiveness, to provide the best possible query results, its internal processing is executed on top of the actual execution of the underlying store. This can have a crucial impact on the entire throughput of the

[9] https://github.com/oltpbenchmark/oltpbench
[10] https://github.com/brianfrankcooper/YCSB



(a) Old Locking

(b) New Locking

(c) Old vs. New Locking on a Single Store

Figure 6.2: Impact of Locking Mechanism on the Overall Throughput per Second.

system.

Hence we used Chronos, a benchmarking tool that allows us to define several evaluation steps per storage engine, to measure the overhead. For this evaluation, the introduced implementation is compared against the current version of Polypheny-DB (v0.7.0). This was executed using a fixed repeating number of operations and measured based on the total execution time that it took for each configuration to apply all steps.

As depicted in Figure 6.1, the results show that the new implementation introduced a little overhead of about 8% for single-store operations. However, with multiple stores the actual execution time was indeed slightly reduced by 5%. Although, single store executions should not be entirely neglected they do not form the main pursuit of Polystore systems, which are more suitably visualized by the multistore runtime.

6.3.2.2 Locking-Mechanism

As described in Section 5.2.2, one of the prerequisites to establish multiple refresh strategies and hence lazy replication, was the refactoring of the locking mechanism. Although, not completely reworked, the locking module of Polyphenys SS2PL poses as a core component of the system. It therefore, impacts correct serializability treatment and is an inherent driver

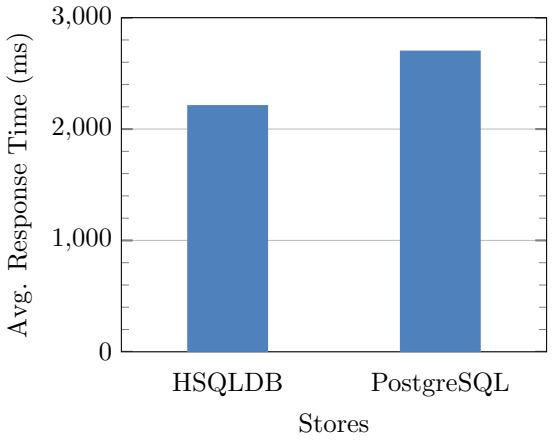
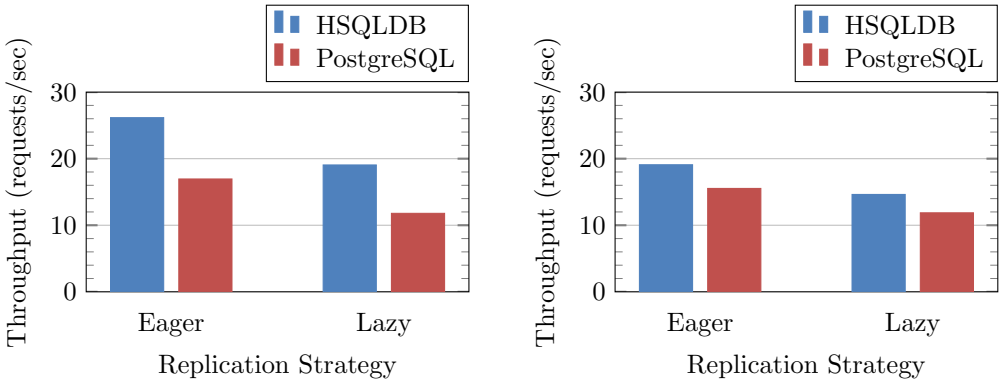


Figure 6.3: Distinct Write Time Comparison of HSQLDB and PostgreSQL.

of the allowed concurrency which directly influences the overall performance of the system. For the evaluation again the current state of Polypheny-DB is compared against this implementation. Since the locking module was changed from a table-wise locking to a partition-wise locking we will validate the impact on the basis of a single table using YCSB. The evaluation was executed with gradually increasing numbers of partitions, which are placed on one store or distributed across n-Stores for n partitions to observe any changes in the locking and therefore the throughput. To get a general overview of the impact, the benchmark was executed using a mixed workload on HSQLDB only. It is evaluated on the number of operations that can be applied to the system per second.
As visualized in Figures 6.2a and 6.2b, for both cases the overall situation is quite similiar. While the distribution of the partitions across several stores gets gradually worse, the single store performance actually improves, the more partitions are added to the table. This behavior is essentially caused by Polyphenys need to join and union several stores together, when querying multiple partitions across several stores. Since more stores need to be connected and considered, it is a rather costly approach and as stated before gets increasingly harder the more stores are involved.
Because the single store variations prove to be more reliable, they are summarized in Figure 6.2c. We can observe that the new locking mechanism, indeed proves to be performing around 9% better in terms of the possible throughput, compared to the old implementation. Furthermore, it shows that again with a growing number of partitions, the gap between the old and new locking extends even more, validating the benefits of the new locking mechanism, if used within strongly partitioned configurations.

6.3.2.3 Baseline Identification

The Lazy Replication algorithm is not only fundamental to generate multiple versions to be used within freshness-awareness, but it is also a core functionality of how data is propagated throughout the system. Hence, along with the newly introduced replication strategies, and *Change Data Collection* it will have a major impact on the overall performance of the system. Since the replication solely focuses on replicating captured changes, the next benchmarks



(a) One process (b) 50 processes

Figure 6.4: Throughput Impact of Concurrency in a Replicated Setup.

will be consequently executed using Write Only operations without any reads.
As stated in the evaluation environment Section 6.3.1, these benchmarks will be mainly executed with an embedded version of HSQLDB and PostgreSQL running within a virtualized container environment. To have a general baseline for comparison Figure 6.3 presents a single store execution, comparing these two stores against each other. As motivated in the beginning, it is crucial for a Polystore system to utilize the key benefits of each store to provide the best results. For our scenario, it is therefore important to determine which store configuration is more suitable to be used as an eagerly replicated primary placement, due to its lower latency and better response time.
This illustrated comparison clearly shows that due to its limited resources the PostgreSQL store performs on average 22% slower and cannot directly compete with this HSQLDB store. This provides us with the intuitive decision to use HSQLDB for the primary transactions.

6.3.2.4 Lazy Replication

As previously stated, the replication strategies will impact the processing capability of the system immensely. A placement with a configured lazy replication strategy automatically enables the system, to start tracking changes for this entity, impacting the duration of a query. Therefore, we want to compare how each store handles the replication. Consequently, we want to benchmark and compare two equal placements that are eagerly replicated against the same two stores but one configured as *lazy*.
To extend the baseline discovered before, we again want to demonstrate the behavior a purely sequential environment with only one client has, against a parallel environment with 50 clients. Figure 6.4 shows the evaluation across two stores, providing the possible throughput per second. This is given as the number of modifications that can be applied to the system per second. As before HSQLDB achieves better results than PostgreSQL. Furthermore, disregarding the underlying store, the eagerly replicated configuration performs much better in all tests, which is not directly apparent when only considering Figure 6.4b. However, considering that the collection of changes within a lazy setup, indeed imposes additional costs on the processing time, such deviations are expected.
Admittingly an entity that is composed of only similar or equal stores, will not be beneficial

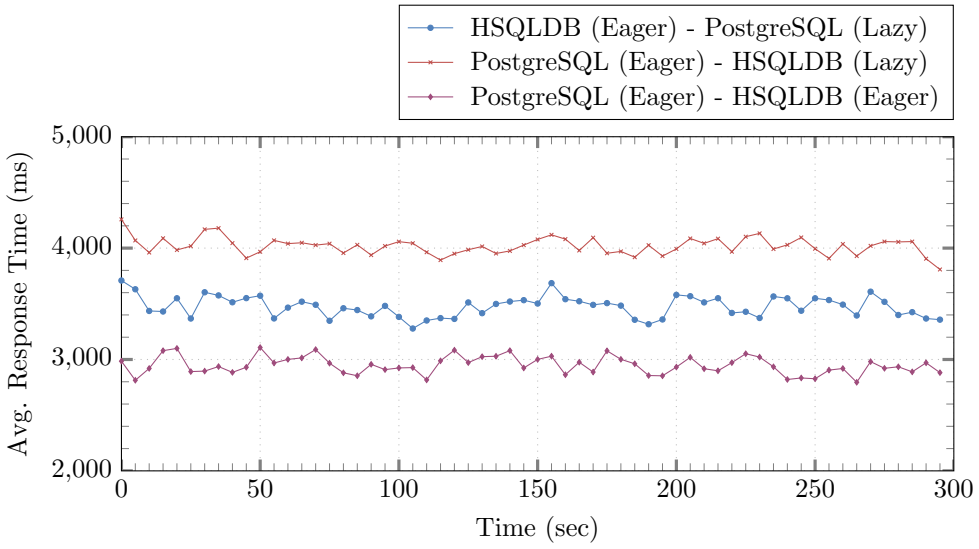


Figure 6.5: Response Time Comparsion of Various Store Configurations – Write Only.

for a Polystore system, to allow different workloads. Therefore the following benchmarks will concentrate on a mixed setup with interleaved stores. These tests will be executed with 50 parallel clients to reproduce a conventional environment.

6.3.2.5 Interleaved Configurations

Now focussing on a mixed setup of two stores containing PostgreSQL as well as HSQLDB for one entity. Nativiely for a Polystore environment we want to identify which setup of underlying stores will produce better results hence is suitable for which situation. Consequently, we want to observe how the order of the stores impacts the response time per benchmark. Again to have a foundation to compare our changes to, we will compare the configurations if both stores are defined as eager and Further respectiviely define each store as lazy as well. Ultimately Figure 6.5 shows that regarding the lazy approaches, we again see the common behavior that HSQLDB performs a little better as an eagerly replicated store compared to PostgreSQL.
Additionally, the Figure in 6.6 puts the average latency in perspective to the execution times described before. As one can observe the eager replications again peform the best while the PostgreSQL variation posing as eager, performs the worst. This not only allows us to compare the different setups but again aids us to choose suitable store combinations to be used for our designated tasks.

However, the presented possibilities so far only considered the execution on two stores. Therefore, Figure 6.7 aims to compare the execution on two and four stores to identify any deviations the lazy replication algorithm has. Again this is done in an interleaved fashion, switching the role of lazy and eager strategy between the participating stores. During these tests only one is eagerly replicated, the remaining stores are all configured as lazy placements. Eagerly and lazily replicated stores in this scenario are defined to be different store types.

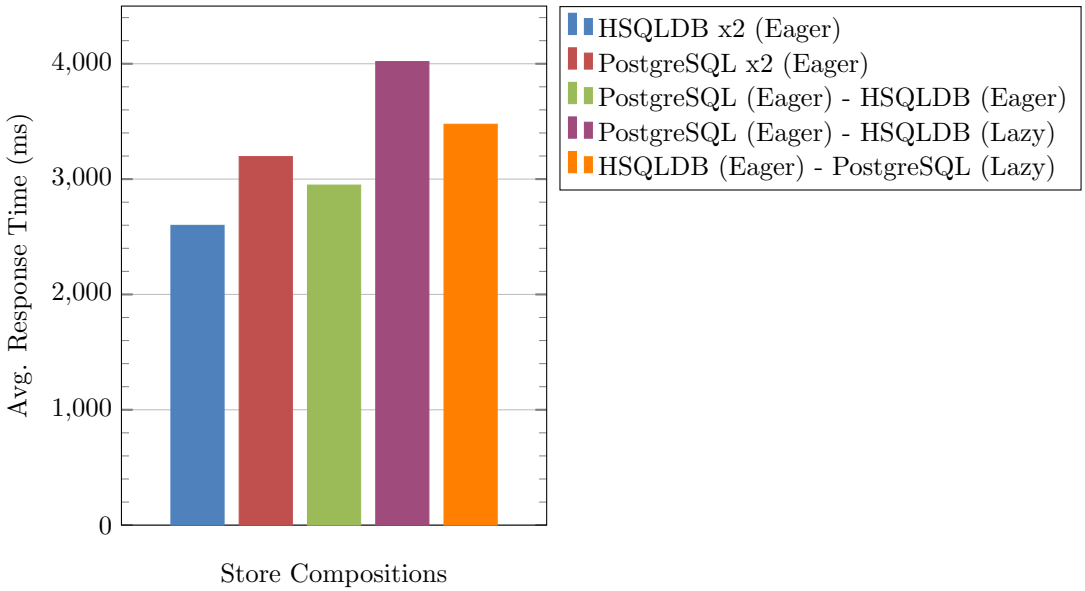
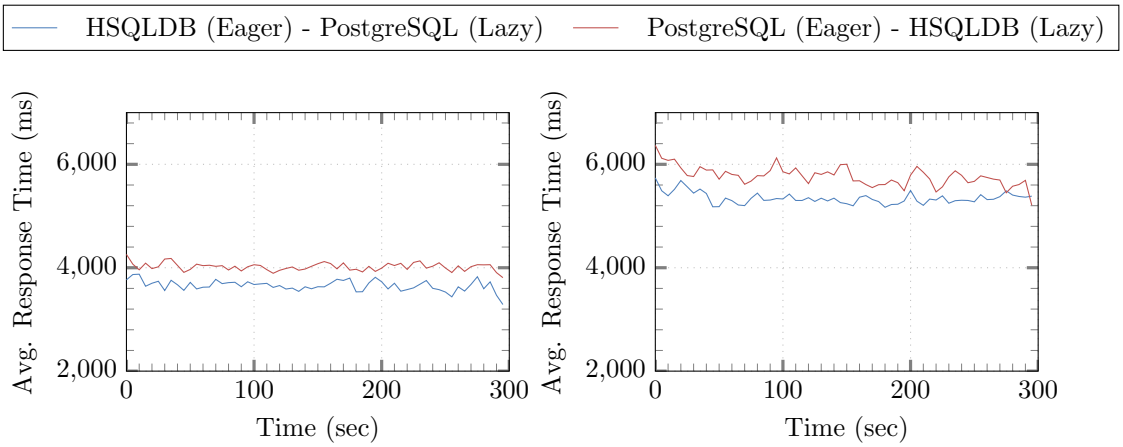


Figure 6.6: Response Time Comparsion of Various Store Compositions – Write Only.

The graphs indicate that although they differ in terms of average response times, the gap between both configurations is comparably equal. In both cases, the execution with the eagerly replicated HSQLDB is roughly 500ms faster in terms of the average response time.

As summarized and visualized more densely in Figure 6.8, its rather counter-intuitive that the configurations in Figure 6.7 (a) and (b) deviate at all. In general, they both only contain one primary placement that is even targeted for the primary transaction. All other stores will be updated asynchronously and are therefore not directly involved. However, as described before, the primary transaction is also responsible for capturing, as well as queuing the changes to be replicated asynchronously. While the procedure is always executed equally, the second approach with four stores has more replication targets that require the change. Since the generation of replication objects as well as the queueing are all still done during the commit of the primary transaction, the observed deviations are reasonable.

Based on this observation we specifically wanted to compare how growth in stores will impact this deviation and how it compares against its eager counterpart. To have a more uniform result this test will be executed using only HSQLDB stores, to have a stable foundation for comparison without a second store that could interfere with the final result.
In this evaluation Figure 6.9 illustrates, the average response time of two to eight stores, where one store is eagerly replicated and the rest is configured as lazy.
This indicates that although the repsonse time increases, it does so in a stable manner. As compared with its eager approach we can see that it evolves roughly towards the same direction plus the previously observed offset, generally providing good scalability.



(a) 2 Stores (b) 4 Stores (Lazy x3)

Figure 6.7: Response Time Comparison Among Different Store Sizes with Interleaved Roles – Write Only.

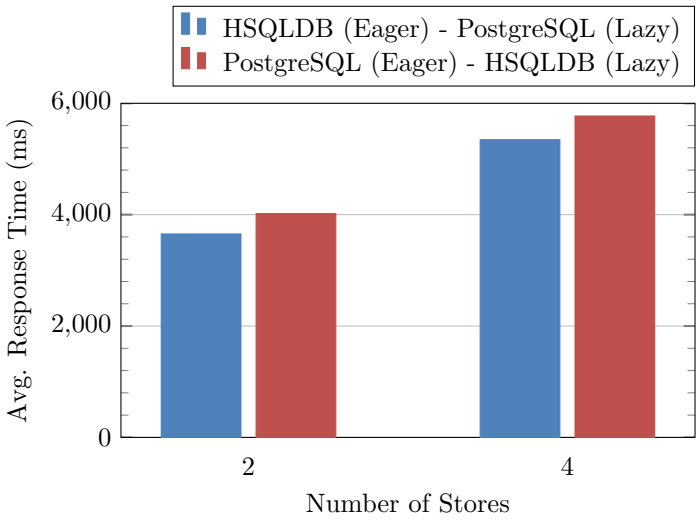


Figure 6.8: Avg. Response Time Comparison of Different Store Sizes (4 Stores → 1x Eager and 3x Lazy) – Write Only.

6.3.2.6 Queue Replication

As presented during the implementation and now elaborated and suggested multiple times during evaluation, the capture of modifications to be propagated to the secondary placements, introduces some overhead. Although some overhead is reasonable due to the additional processing steps, the response times differ especially when compared to regular eagerly replicated scenarios. This directly impacts the overall performance, resulting in generally higher response times for lazy replication scenarios. Therefore we want to identify for a single write-operation what actually influences these runtime deviations.

The comparison in Figure 6.10 analyzes the execution time of two independent write operations, respectively executed on two equal stores. One configuration is a simple eager replicated entity on two stores without any replication, the other one is an operation that needs to capture the modification within the global replication queue.

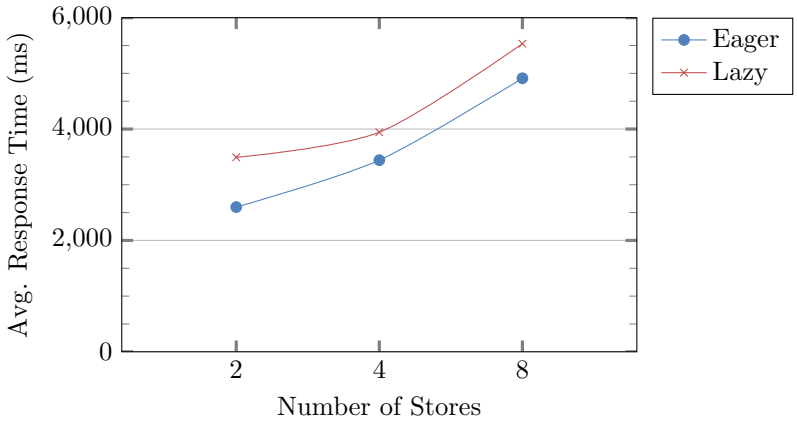


Figure 6.9: Replication Strategy Comparison with Increasing Stores – Write Only.

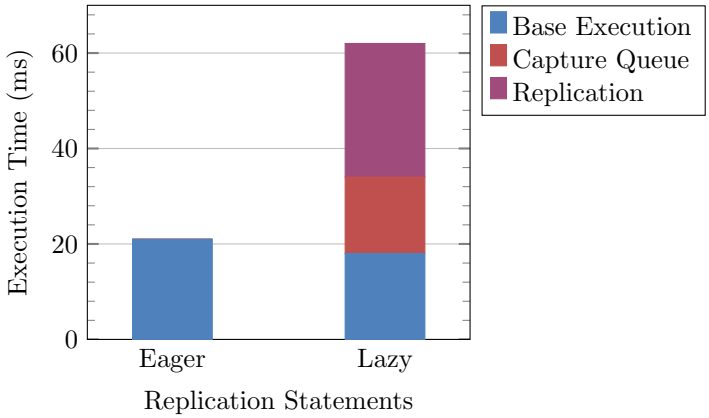


Figure 6.10: Execution Time Comparision of a Decompossed Distinct Write-Operation with and without Active Data Capture.

Since the lazy approach only needs to involve one store for processing, the baseline of the lazy approach is indeed slightly faster than its eager counterpart. However, considering that at commit time this approach also needs to extract the captured change, convert it into a replication object and ultimately queue it for the remaining store, the actual execution deviates quite heavily. With the queue time being almost equal to its base execution time. However, since the capture part is only executed once at the commit time of a transaction, it drastically impacts the performance of one single operation. As we have seen before, this introduced capture gap remains quite stable even with additional targets or capture objects to transform. Therefore the queueing process negatively impacts the efficiency of transactions containing only one or few operations. In contrast, it gets neglected further the more operations are executed within one transaction.

Additionally, although not directly considered to be part of the execution time, is the convergence window. As mentioned before, as soon as a replication worker will have free resources it will replicate the pending object from the queue onto the secondary store. As we can see the replication duration takes a little bit longer than the actual execution. This is caused by the introduced validation constraints for each worker to assess the queue and reconstruct each operation individually. However, since the replication is done operation-wise the repli-

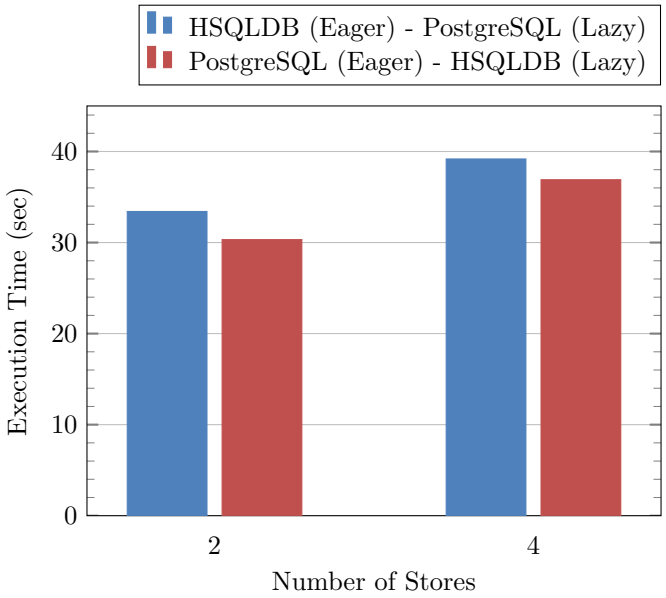


Figure 6.11: Convergence Time Comparison on Different Store Constellations and Sizes (4 Stores → 1x Eager and 3x Lazy) – Mixed Workload.

cation time is considered to be fixed per operation. Therefore, the entire bar reflects the time it takes for one operation to be executed and replicated across all participating stores, considering the aforementioned constraints of the queue time. Since the eagerly replicated approach is executed within one single transaction targeting both underlying stores, the entity is considered to be immediately consistent and will not need to converge.

6.3.2.7 Replica Convergence

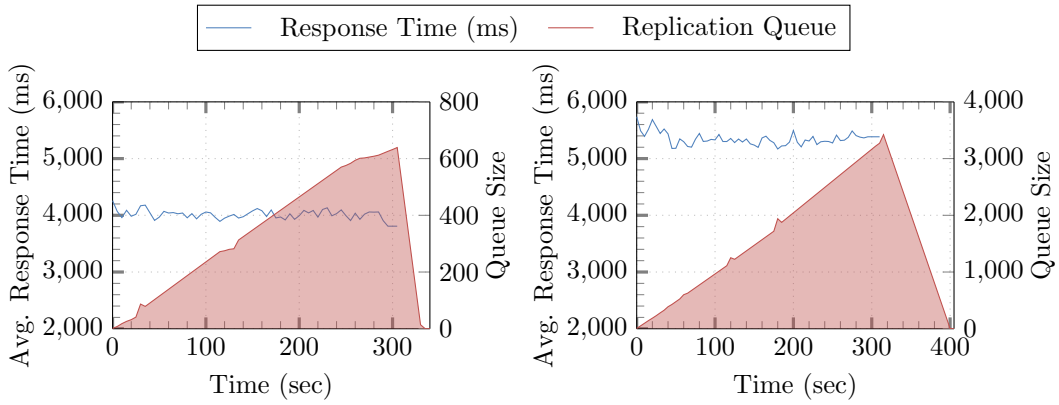
As we have seen before the versions can deviate quite heavily during a mixed workload. Furthermore, we have already identified that configuring the faster performing store to be replicated eagerly, prooves that this will positively impact the availability due to its lower response time of the initial transaction. However as important as the throughput of the initial transaction might be, choosing the faster node to be eagerly replicated intuitively leaves the slower node to apply the updates asynchronously. However, because it highly depends on the requirements of which configuration to choose, we also want to establish a setup that focuses on quickly reaching a consistent state without using an eager-only replication setup.

Therefore, we want to observe the actual replication time until the two stores reach an equilibrium in terms of their received updates.

The plot in Figure 6.11 visualizes the comparison based on a different number of stores.

Due to the fact that HSQLDB was before identified as the store with the better throughput, it also generally manages to converge faster if configured to be the secondary store. Since HSQLDB in our scenario can replicate the operations faster than the primary transaction can queue new events, it does not only converge faster but has a lower execution time in general.

Therefore we have established that with our introduced algorithm, indeed the better per-



(a) 2 Stores (b) 4 Stores

Figure 6.12: Execution Time along the Replication Queue Convergence – Mixed Workload.

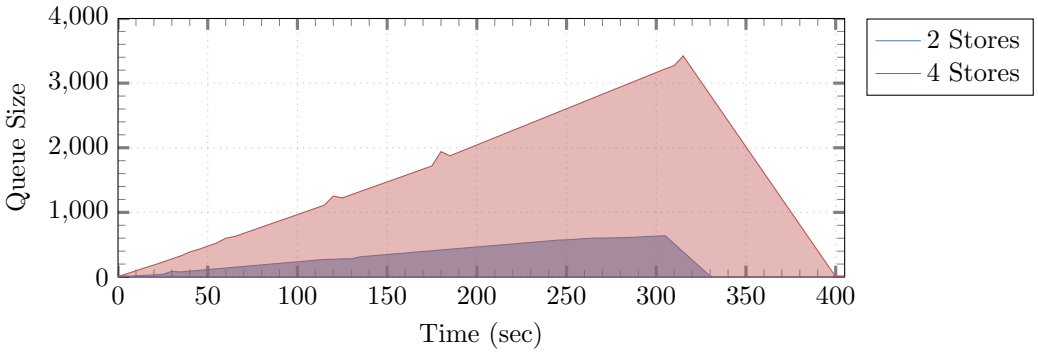


Figure 6.13: Replication Queue Convergence Over Time – Mixed Workload.

forming store is also more suitable when we want to reach consistency faster, hence providing a smaller convergence time.

Although the convergence time is not essentially impacted by the executing store, but also by the number of designated stores that shall asynchronously receive the operation. Further, we want to identify how the queue behaves if the secondary system is generally slower and therefore not able to replicate the pending changes as fast as new ones are added to the queue.

For this test, we are using a Mixed Workload to allow permanent access to the primary node (HSQLDB), while the lazy replicated nodes (PostgreSQL) can solely concentrate on applying the queued changes. Figure 6.12 visualizes the expansion- as well as the shrinking-phase of the replication queue and puts it into perspective along with the actual test execution. During these tests, one worker continuously processed the queue and replicated the changes operation-wise. As illustrated in both cases the worker is not able to apply the changes faster than they are coming in. The breakeven point is reached as soon as the test execution has finished and the load on the system stops. For both configurations, the algorithm can now quickly propagate all pending changes.

As mentioned before the accumulation of replication events is essentially caused by the

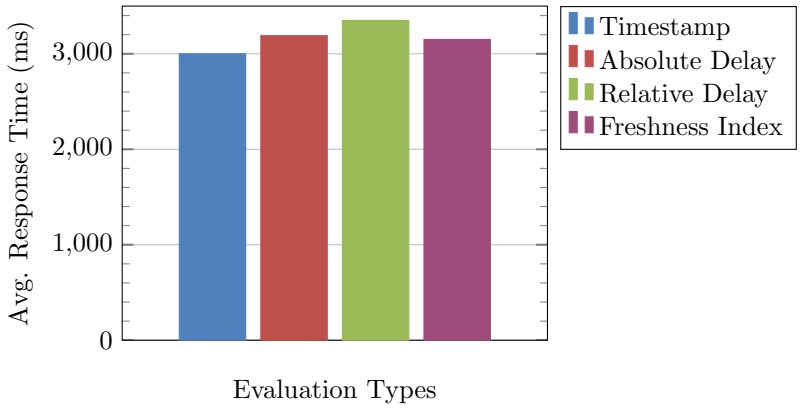


Figure 6.14: Avg. Response Time Comparison of Available Evaluation Types – Freshness Only

different performance capabilities of the underlying stores. While HSQLDB as a lazy replicated store would process and apply the events faster, leaving the queue mostly empty, PostgreSQL is not able to match this performance by applying the replication events slower than new ones are generated. This then causes the queue to grow until there is no more load on the system. However, without a defined test enviornment in regular situations with continuous load, this could cause severe problems, since the queue will grow without bound. Because captured change operations are also transformed $1 - n$ for n lazy replicated stores, the size of the queues is further influenced by the number of replicas that require the change. As summarized in Figure 6.13 this directly influences the earlier described convergence time of our system and negatively impacts the response time. While the configuration with two stores only needs an additional 30 seconds (10% of total execution time) to converge, a setup with four stores already needs 105 seconds (35%) to converge the remaining stores. Which makes the selected storage constellation crucial.

6.3.2.8 Freshness Evaluation Type Filter

The benchmarks so far merely focussed on the replication as well as the write-operations. Therefore the next evaluations will concentrate on testing the introduced freshness capabilities.

Despite the actual usage of freshness operations in general, also the chosen evaluation type will influence the performance. Although, that the filter comparison itself is always executed equally, the filter generation can differ between the available types.

While a timestamp can be directly used as it is, an index first needs to aggregate the total number of modifications per placement and calculate the comparison-index based on the deviation from the master. Albeit, that this is not a very costly operation, it will have a significant impact when executed multiple times. To obtain comparable results, the benchmarks have been executed with the most liberal degree of freshness per type, to avoid any side effects or fallbacks to the primary nodes.

Figure 6.14 therefore presents the overall latency of a mixed workload accompanied by the different evaluation types. As already suggested all types provide a very similar response

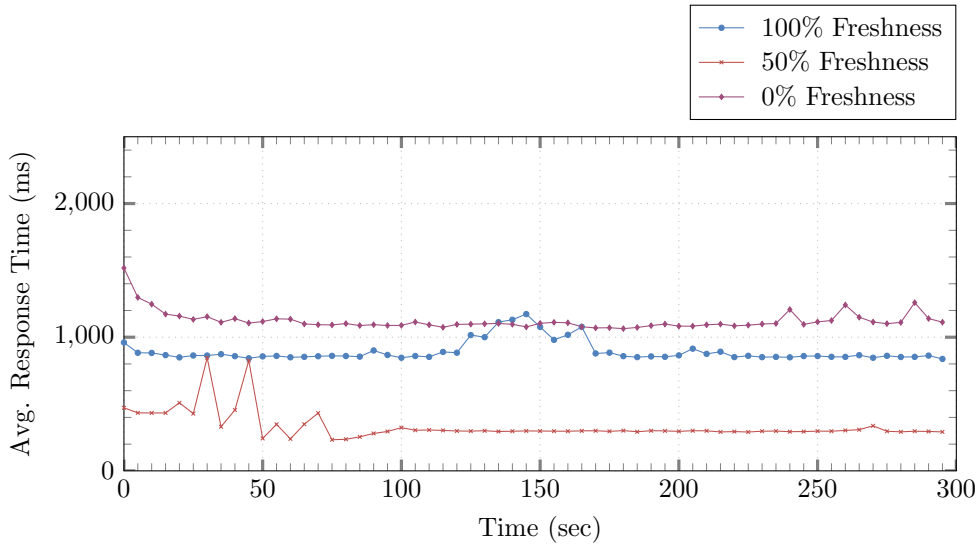


Figure 6.15: Response Time Comparision on Varying Proportions of Freshness.

time. A difference becomes only really obvious when e.g. comparing a natively usable timestamp to a relative time delay. While the first one can be applied directly the latter first needs to extract the commit time of all possible placements and apply the deviation before it is able to filter. This plot shows that although only marginally different the utilized filter will still slightly impact the overall performance.

6.3.2.9 Freshness-Aware Read Operations

As with write-only operations also the freshness can directly indicate a suitable store combination. While write-operations focus on a trade-off between latency of the primary transaction and a faster convergence time, the read-operations will consequently focus on the target where the read is applied. While regular read-operations exclusively target primary placements, freshness-queries will mainly be redirected towards possibly outdated secondary replicas.

As described before, since freshness-related results are essentially influenced by the entity composition, we need an independent measurement. Therefore we will compare two equal stores (HSQLDB) to again allow a base comparison and observe how the freshness specification itself impacts the results.

Therefore Figure 6.15 shows a comparison of different degrees of freshness used within a pure read-only environment. These results generally show that the utilized freshness alone is already sufficient to provide better read results than if no freshness is used at all. As before this is inherently caused by the targeted store of the query. While all queries without a freshness specification will directly contact a primary node, hence needing to apply a lock, freshness queries can contact the secondary node without a lock. Although in general shared-locks can be easily applied for regular read-operations, they still need to be formally acquired impacting the throughput. The most benefit can therefore be gained by combining the freshness with regular operations. As depicted above, these combinations provide by far the best results. Despite that regular reads still need to acquire a lock, these different query

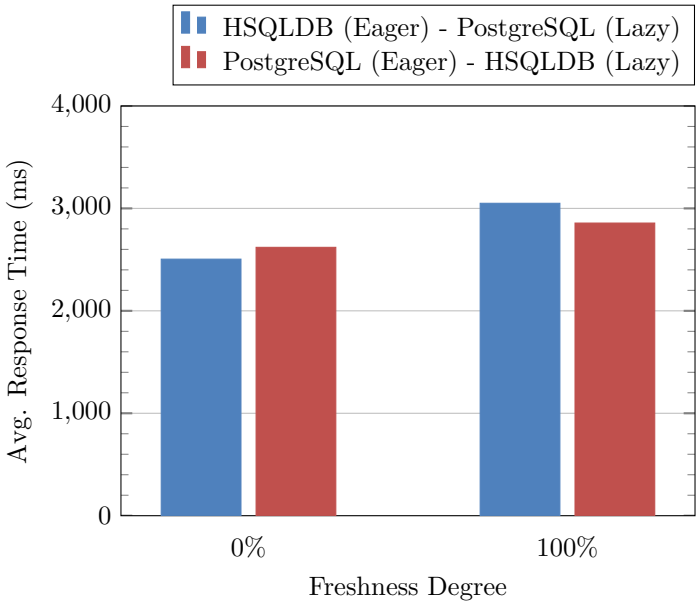


Figure 6.16: Impact of Store Selection based on Varying Proportions of Freshness – Partial Freshness.

types will consequently also target different stores. This allows an efficient load distribution across the utilized stores, to harvest the benefits of a distributed environment increasing the overall throughput.

Even with a mixed workload without any freshness constraints we already observed that the store constellation does have an impact on the overall latency and influences the decision process. These effects can also be considered when evaluating a partial workload on our interleaved setup with HSQLDB and PostgreSQL.

The illustration in 6.16, therefore indicates the impact a given store constellation has on the overall throughput. While regular operations perform better in the configuration targeting HSQLDB as the primary node, whereas queries with freshness, target the secondary node essentially flipping the roles. This is again caused by the target of the select statements. Since we are rather in a read-heavy environment the selection has quite the impact on the final result. This proves that the store constellation will strongly influence the systems behavior as well as the designated workloads to use this constellation on.

6.3.2.10 Compound Operations

Now with every building block evaluated independently, we will combine and assess the given functionality jointly.

Based on the elementary comparison in Figure 6.10 we have already established the heavy impact the capture-queue as well as the replication have on a single write-operation.

As mentioned in the implementation, depending on the current load on the system, we might observe contentions since the replication is done in parallel. Furthermore, we might even come across situations where an administrator will directly define a placement as *INFINITELY OUTDATED* to not receive any more updates and retain its current version.

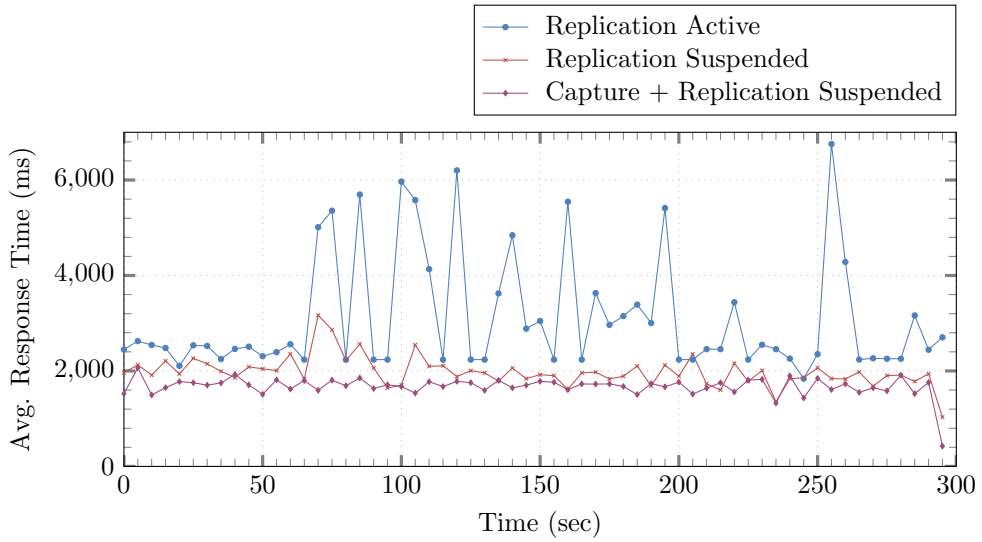


Figure 6.17: Impact of Replication Queue on Response Time using Freshness (Index=0.7) – Partial Freshness.

For the next benchmark, we therefore, want to see how the workload behaves if the replication is done in parallel if the replication distribution is suspended and if the entire queue-capture is suspended. This is again done using two HSQLDB stores, to reduce possible deviations across stores. Additionally, this benchmark introduces freshness queries, to observe how it influences the overall response time even when there is no active replication. The freshness evaluation utilizes a freshness-index of 0.7 to filter and identify possible candidates based on the number of received modifications.

As provided by Figure 6.17 we can again observe that the overall response time improves if there is no active replication running in the background. Furthermore, the system performs best if it does not need to capture any changes at all, which essentially corresponds to the observations we have described in Figure 6.10. Although this is rather a corner case, the replication suspension on the other hand is more likely to occur and still provide good response times.

Interestingly, we further observe peaks on the active replication graph. While the graphs without replication and capture mechanisms are more narrow, the active replication queue oscillates quite heavily, negatively impacting the average response time.

Further, we observe that these peaks start occurring after roughly one minute. Since we used a modification deviation for our freshness specification, the system can quickly identify suitable placements at the beginning of the benchmark. However, after some time the versions start deviating quite heavily from their eager version. Since the updates are only propagated operation-wise, the replications cannot catch up with the primary operations anymore as already presented in Figure Referencesfig:converge24. Hence the system is not able to fulfill the tolerated freshness-level and causes the algorithm to fall back to its primary placement, again investing additional time to combine and select a suitable combination of placements, which ultimately causes the peaks.

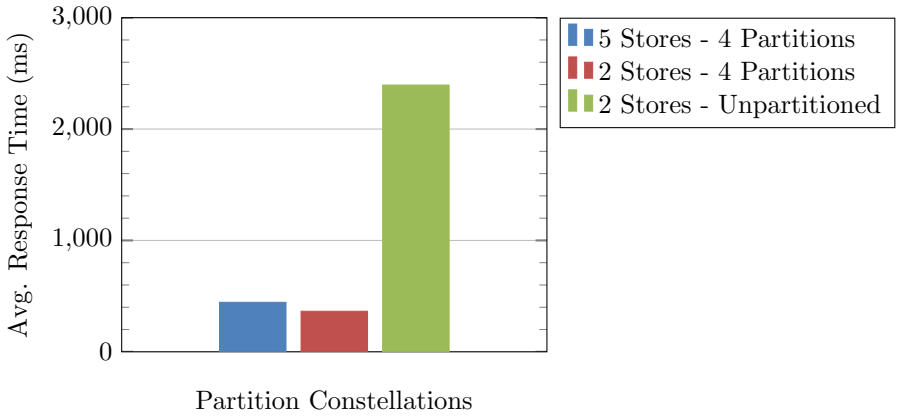


Figure 6.18: Impact of Partitioning on Freshness (Index=0.7), Across Lazy Replicated Stores – Partial Freshness.

6.3.2.11 Partitioning

Our freshness and replication implementation essentially revolves around the handling of individual Partition Placements. However, so far all tests only really considered unpartitioned entities to visualize basic differences.

Accompanied by the introduced locking changes and the promising results described in Figure 6.2, we will evaluate the Partial Freshness workload in a partitioned environment. For one we will partition the table into four horizontal partitions. For one scenario we place all partitions on the eager node and then each partition respectively on four lazy replicated stores to achieve a highly distributed setup as before. The second scenario will place all four partitions across two stores, where exactly one is eagerly and the other lazily replicated. These two variations are ultimately compared against an entire unpartitioned setup across two HSQLDB stores which is visualized in Figure 6.18.

Analogously, to the results obtained from the locking mechanism benchmarks, these results consequently indicate, that we achieve the best performance if we indeed partition the entity and place all partitions on each entity across both stores. This natively allows the system to have the most possible flexibility in locking and accessing the individual data fragments. Ultimately this does not only mimic the performance gains we have already observed due to the locking but also extends it even further when ingesting freshness constraints.

6.4 Workload Comparison

Concluding with the evaluations and as illustrated with the last examples and benchmarks, the results of freshness constraints, inherently differs through the utilized workload. On the basis of fully freshness optimized read-operations using an index of 0.7, and two HSQLDB stores, Figure 6.19 summarizes the impact of the individual workload classes on freshness-aware data management in general.

As we have identified before the most extreme cases like read-only and write-only workloads achieve the best results, since there are no negatively impacting concurrent operations.

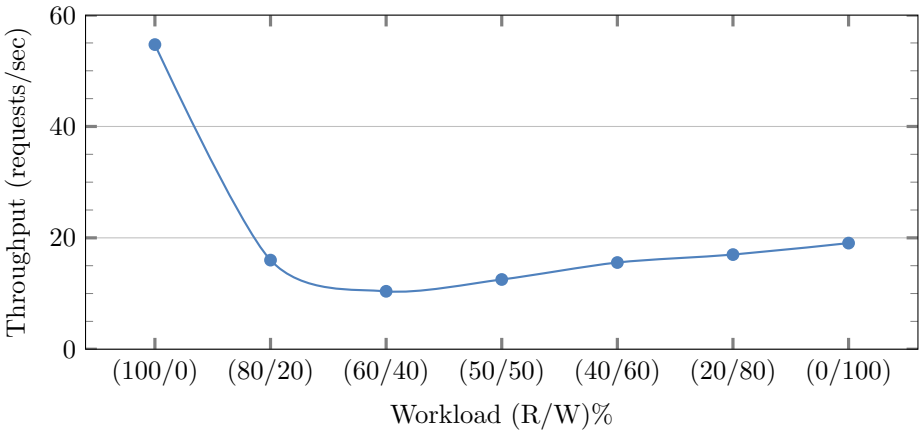


Figure 6.19: Workload Comparison of Freshness Related Queries (Index=0.7).

However, because these are not considered to be common workloads, the next best operation is considered to be a write-heavy workload which only partially focuses on reads. This essentially mimics a highly transactional workload using the reads mainly for analytical purposes.
This shows that freshness-aware data management can indeed improve mixed workloads during write-heavy situations.

6.5 Discussion

The results generally show that although the replication will slightly mitigate the overall performance of the system, it still can correctly fulfill the requirements without largely interfering with the underlying systems. Despite that the operation-wise execution will gradually progress each placement as intended, the capture-queue inherently impacts the performance of parallel workload when using a lazy replicated approach.
While eagerly replicated setups generally performed better in write-heavy environments, the compound results still show that freshness related queries are indeed suitable candidates when not focusing on a pure mixed-workload with equal amounts of read- and write-operations. Especially the partitioned cases showed promising results to optimize routing decisions to finally improve the overall performance.
Although the freshness queries already provided good results, they still suffered from a rather static freshness specification. Therefore it would make sense to extend and adapt the utilized benchmarks to adaptively adjust the degree of freshness during runtime as needed and still providing reproducible results.
So far we only focused on the basic functionalities and suitable fields of application. Since also the respective freshness degree will impact the choice of which of the proposed candidates fulfills the requirements of the query, it would make sense to further benchmark different varying freshness constraints to observe their long-term behavior.

7 Conclusion

In this thesis, we have presented a concept as well as a solution for freshness-aware data management within a Polystore system. With this implementation, we have introduced a possibility to allow system administrators, or operators in general, to define their replication requirements as needed. With the established replication strategies and states, we can define on a table-level basis how updates are propagated within the system and can therefore directly influence the availability and consistency per object.
Based on the established functional requirements for data freshness, we have presented metrics, that us allow to flexibly define a tolerated level of freshness, depending on varying use cases and scenarios. This immediately enables us to use possibly outdated nodes, containing stale data, to be utilized during retrieval to support analytical queries even in the presence of transactional workload. This does not only improve parallel processing in general, but also allows to leverage the inherent redundancy of a PolyDBMS to efficiently utilize the key benefits of the underlying stores.
Despite that, the performance, as well as the convergence time, is better within the eagerly replicated case, the lazy approach allows to efficiently utilize all remaining resources. Further, we have introduced a reliable fault-tolerant replication algorithm, which can be used to lazily replicate changes to outdated replicas. It ensures the consistency of each placement by enforcing the natural execution order, while automatically rescheduling failed replications and removing remaining replications from suspended or removed placements, hence cleansing the environment at runtime.

Depending on the workload, the system is however still able to provide great results due to the introduced nuances of freshness to support various scenarios. Albeit not being able to support all workloads equally, the implementation still offers a broad field of application. Polypheny-DB therefore now provides functionalities to adjust itself to the concepts revolving around *CAP* and *PACELC*, described in Section 3.2, to let users choose between consistency and availability, by decoupling primary and secondary updates and deferring refresh operations to a later point in time. Due to this asynchrony, it now efficiently supports hybrid workload. Together with the adjusted locking capabilities, it can achieve great results, utilizing any freshness constraint in a partitioned setup. It now provides underlying

systems with the opportunity to accept freshness queries, to efficiently utilize all available resources of the entire system.
Since our freshness filter is analyzed and evaluated centrally within the Polystore layer, it can be applied to arbitrary underlying stores. When the stores are partially replicated, several of the underlying stores may qualify for the execution of these queries. In order to avoid that single stores are overloaded, the query processing and optimization steps can effectively consider version selection and load balance the access among relevant replicas. For this reason, our approach further allows comparing different placements, disregarding their actual physical representation. We can therefore compare them simply on the basis of their update information despite that they might contain different column placements. This again allows using the Polystore system to dynamically process the freshness specification independently.
In general, we can even reduce the data footprint since we are certain, that we do not have infinitely available versions as in conventional multi-version database systems. This enables a freshness specification without any determined tolerated level and promises a certain level of freshness by design. Although not all entities have to support freshness users can act agnostic of the underlying architecture and specify the freshness either way, since we can always fallback and route the query towards a primary placement.

Although the capture-queue currently negatively impacts the overall performance of the system, it could be easily improved by deferring the creation of replication objects from the commit time. Instead of creating the designated target replications within the primary transaction, we can pass them to an intermediate process, which will transform the change objects into replication events for us. Additionally, this can be further optimized by investigating the queue periodically to refrain from replicating operation-wise. The system could therefore be easily adjusted to inspect the queue first, then try to aggregate as many subsequent operations as possible and replicate them jointly.
Of course, there are more possibilities to improve the system, such as the queue's persistency. While enabling fast access when storing replication in memory it is prone to failures and outages, losing relevant data to be replicated. In order to remove the workarounds during start-up, the replication queues, as well as the data, have to be persisted. Despite slower processing times for primary update transactions, we would increase the stability and the recoverability of the system immensely. In the end this again is a trade-off between availability and consistency, depending on individual requirements. This is however an obvious extension of this work and introduces a foundation for future work.

8 Outlook

The introduction of freshness within a PolyDBMS generally allows utilizing the different capabilities of the underlying stores and harvest their individual benefits, while now being able to trade between availability and consistency. This allows to scale Polystores globally and use them to fulfill an even broader range of new requirements.
The introduced implementation therefore already offers a great foundation for additional future work. Despite the proposed improvements of the current implementation, the notion of freshness can be extended even further to provide more dynamic approaches, moving the system toward a self-adjustable environment.

8.1 Tunable Consistency

The introduced implementation sketched in Section 5 reduces the overall consistency of the primary transaction, to improve the overall response time of the system.
But since this trade-off between availability and consistency certainly depends on the use case or service requirements, it would be beneficial to let the system adapt itself based on centrally defined policy constraints. Such policies could e.g. define how many replicas need to be maintained within a specific schema to remain available or further allow to define a minimum number of stores that always need to apply an update, to ensure its consistency. Instead of labeling fixed data placements to receive updates eagerly, we could allow a more flexible approach that is sufficient if already a single placement responds to the update, disregarding its role. The predefined replication state can be therefore omitted. Additionally, this can also be adjusted with general freshness requirements, to automatically maintain different designated versions of data objects without having an administrator, manually managing the system. Hence, an extension to the described model could easily allow adjusting the required consistency as needed. Such approaches can then be easily combined with tunable consistency to allow self-adjusting data placements adapting to individual use cases.

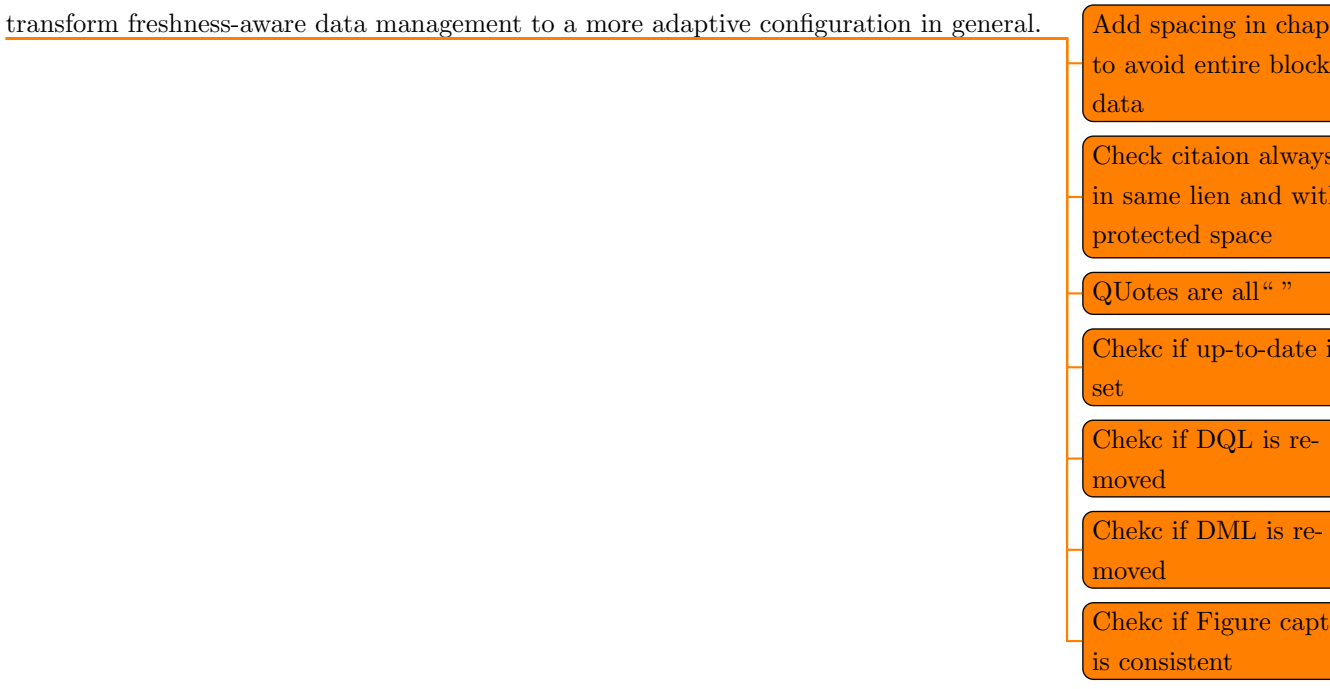
8.2 Global Replication Strategies

This implementation has only introduced the specification of table-level entities like entire data placements to be defined as eagerly or lazily replicated objects. Although this intro-

duces a high degree of flexibility, it still might be desirable to define certain policies, so that entire schemas or even databases automatically receive updates lazily, while still ensuring the overall placement constraints.
This concept could be extended even further by applying it to a distributed setup of Polypheny-DB, that replicates data autonomously to certain regions based on the given configuration. This extension could leverage the introduced freshness-awareness to consider off-site locations to be used for even more parallel workload. This could even include using a more adaptive freshness approach that evaluates the distance to a node to identify suitable locations for retrieval. Since outdated information fetched from a distant store, will become more outdated during transfer, we can extend this further to even measure the common round trip time to that store and integrate it into the query specification to identify suitable placements.

8.3 Adaptive Session-Wide Freshness

Another addition to freshness could be the extension to also allow the specification of freshness per session, which avoids specifying the freshness for individual statements. This can be especially useful if the freshness requirements do not change over time, allowing a quick possibility to manifest the requirements. Although they could be extended for individual statements, that indeed require a more strict form of freshness, it provides a good baseline to operate using freshness. This is especially interesting for applications that usually establish one session, for the majority of their lifetime. Such applications currently have to internally define per statement what degree of freshness they might tolerate. A session-wide freshness could therefore allow centrally adapting the freshness if the requirement changes, without having to recompile the application again. Together with adjustable freshness configurations that might automatically adapt towards recently provided freshness data, it would allow to transform freshness-aware data management to a more adaptive configuration in general.



Bibliography

[1] D. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42, 2012. doi: 10.1109/MC.2012.33.

[2] J. Abadi, R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 967–980. Association for Computing Machinery, 2008. doi: 10.1145/1376616.1376712.

[3] R. Agrawal, A. Ailamaki, P. Bernstein, E. Brewer, M. Carey, and S. Chaudhuri. The Claremont Report on Database Research. *Commun. ACM*, 52(6):56–65, 2009. doi: 10.1145/1516046.1516062.

[4] S. Agrawal, V. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 359–370. Association for Computing Machinery, 2004. doi: 10.1145/1007568.1007609.

[5] F. Akal, C. Türker, H. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. *VLDB 2005 - Proceedings of 31st International Conference on Very Large Data Bases*, 2:465 – 576, 2005.

[6] A. Bedewy, Y. Sun, and N. Shroff. Optimizing data freshness, throughput, and delay in multi-server information-update systems. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 2569–2573, 2016. doi: 10.1109/ISIT.2016.7541763.

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.*, 24(2):1–10, 1995. doi: 10.1145/568271.223785.

[8] P. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. volume 13, page 185–221. Association for Computing Machinery, 1981. doi: 10.1145/356842.356846.

[9] P. Bernstein and N. Goodman. Concurrency Control Algorithms for Multiversion Database Systems. In *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '82, page 209–215. Association for Computing Machinery, 1982. doi: 10.1145/800220.806699.

[10] P. Bernstein and N. Goodman. Multiversion Concurrency Control—Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983. doi: 10.1145/319996.319998.

[11] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1987. ISBN 0-201-10715-5.

[12] E. Brewer. Towards robust distributed systems. PODC '00. Association for Computing Machinery, 2000. doi: 10.1145/343477.343502.

[13] F. Brinkmann and H. Schuldt. Towards Archiving-as-a-Service: A Distributed Index for the Cost-Effective Access to Replicated Multi-Version Data. IDEAS '15, page 81–89. Association for Computing Machinery, 2015. doi: 10.1145/2790755.2790770.

[14] L. Campbell and C. Majors. *Database Reliability Engineering*. O'Reilly, 2017. ISBN 978-1-491-92594-2.

[15] S. Ceri, M. Negri, and G. Pelagatti. Horizontal Data Partitioning in Database Design. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, SIGMOD '82, page 128–136. Association for Computing Machinery, 1982. doi: 10.1145/582353.582376.

[16] J. Cho and H. Garcia-Molina. Synchronizing a Database to Improve Freshness. *SIGMOD Rec.*, 29(2):117–128, 2000. doi: 10.1145/335191.335391.

[17] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154. Association for Computing Machinery, 2010. doi: 10.1145/1807128.1807152.

[18] H. Darwen, C. Date, and R. Fagin. A Normal Form for Preventing Redundant Tuples in Relational Databases. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, page 114–126. Association for Computing Machinery, 2012. doi: 10.1145/2274576.2274589.

[19] S. Dasgupta, K. Coakley, and A. Gupta. Analytics-driven data ingestion and derivation in the AWESOME polystore. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2555–2564, 2016. doi: 10.1109/BigData.2016.7840897.

[20] K. Daudjee and K. Salem. Lazy Database Replication with Snapshot Isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 715–726, 2006.

[21] D. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, 2013. doi: 10.14778/2732240.2732246.

[22] O. Etzion, S. Jajodia, and S. Sripada, editors. *Temporal Databases: Research and Practice*. Springer, 1998. ISBN 978-3-540-69799-1.

[23] J. Faleiro and D. Abadi. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.*, 8(11):1190–1201, 2015. doi: 10.14778/2809974.2809981.

[24] F. Färber, S. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4): 45–51, 2012. doi: 10.1145/2094114.2094126.

[25] A. Fekete. Replica Freshness. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. doi: 10.1007/978-1-4614-8265-9.

[26] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005. doi: 10.1145/1071610.1071615.

[27] M. Fowler. Polyglot persistence, 2011 (accessed May 04, 2022). URL https://martinfowler.com/bliki/PolyglotPersistence.html.

[28] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002. doi: 10.1145/564585.564601.

[29] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 173–182. Association for Computing Machinery, 1996. doi: 10.1145/233269.233330.

[30] M. Hennemann. *Freshness-aware Data Management in a Polystore System*. Project report, University of Basel, December 2021.

[31] C. Huang, M. Cahill, A. Fekete, and Uwe Röhm. Deciding when to trade data freshness for performance in mongodb-as-a-service. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1934–1937, 2020. doi: 10.1109/ICDE48307.2020.00207.

[32] R. Jiménez-Peris, M. Patiño Martínez, G. Alonso, and B. Kemme. Are Quorums an Alternative for Data Replication? *ACM Trans. Database Syst.*, 28(3):257–294, 2003. doi: 10.1145/937598.937601.

[33] J. Levandoski, P. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, ICDE '13, pages 26–37. IEEE Computer Society, 2013. doi: 10.1109/ICDE.2013.6544811.

[34] S. Mehrotra, H. Korth, and A. Silberschatz. Concurrency Control in Hierarchical Multidatabase Systems. *VLDB*, 6(2):152–172, 1997. doi: 10.1007/s007780050038.

[35] F. Naumann, U. Leser, and J. Freytag. Quality-driven integration of heterogeneous information systems. VLDB '99, page 447–458. Morgan Kaufmann Publishers Inc., 1999. ISBN 1558606157.

[36] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, 1984. doi: 10.1145/1994.2209.

[37] E. Pacitti and E. Simon. Update Propagation Strategies to Improve Freshness in Lazy Master Replicated Databases. *The VLDB Journal*, 8:305–318, 2000. doi: 10.1007/s007780050010.

[38] E. Pacitti, C. Coulon, , and P. Valduriez. Preventive replication in a database cluster. volume 18, page 223–251, 2005. doi: 10.1007/s10619-005-4257-4.

[39] V. Peralta, R. Ruggia, and M. Bouzeghoub. Analyzing and Evaluating Data Freshness in Data Integration Systems. *Ingénierie des Systèmes d'Information*, 9:145–162, 2004. doi: 10.3166/isi.9.5-6.145-162.

[40] H. Plattner and B. Leukert. *The In-Memory Revolution*. Springer, 1 edition, 2015. ISBN 978-3-319-16672-8.

[41] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K. Sattler. Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. Springer International Publishing, 2014. doi: 10.1007/978-3-319-15350-6_7.

[42] T. Redman and A. Godfrey. *Data Quality for the Information Age*. Artech House, Inc., 1997. ISBN 0890068836.

[43] U. Röhm, K. Böhm, Schek K., and H. Schuldt. FAS - A freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*, pages 754–765. Morgan Kaufmann, 2002. doi: 10.1016/B978-155860869-6/50072-X.

[44] M. Shapiro. A Principled Approach to Eventual Consistency. In *2011 IEEE 20th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE Computer Society, 2011. doi: 10.1109/WETICE.2011.76.

[45] M. Stonebraker and U. Cetintemel. "One size fits all": an idea whose time has come and gone. In *21st International Conference on Data Engineering (ICDE'05)*, pages 2–11, 2005. doi: 10.1109/ICDE.2005.1.

[46] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, and M. Ferreira. C-store: a column-oriented DBMS. volume 2 of *VLDB '05*, pages 553–564. VLDB Endowment, 2005. ISBN 1595931546.

[47] D. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 309–324. Association for Computing Machinery, 2013. doi: 10.1145/2517349.2522731.

[48] M. Vogt. *Adaptive Management of Multimodel Data and Heterogeneous Workloads*. Dissertation, University of Basel, 2022.

[49] M. Vogt, A. Stiemer, and H. Schuldt. Polypheny-db: Towards a distributed and self-adaptive polystore. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3364–3373, 2018. doi: 10.1109/BigData.2018.8622353.

[50] M. Vogt, N. Hansen, J. Schönholz, D. Lengweiler, I. Geissmann, S. Philipp, Stiemer A., and H. Schuldt. Polypheny-db: Towards bridging the gap between polystores and htap systems. In *Proceedings of the 3rd International Workshop on Polystore systems for heterogeneous data in multiple databases with privacy and security assurance (Poly' 2020)*, page 25–36, 2020. doi: 10.1007/978-3-030-71055-2_2.

[51] M. Vogt, A. Stiemer, S. Coray, and H. Schuldt. Chronos: The swiss army knife for database evaluations. In *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pages 583–586. OpenProceedings.org, 2020. doi: 10.5441/002/edbt.2020.69.

[52] M. Vogt, D. Lengweiler, I. Geissmann, N. Hansen, M. Hennemann, C. Mendelin, S. Philipp, and H. Schuldt. Polystore Systems And DBMSs: Love Marriage Or Marriage Of Convenience? In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare: VLDB Workshops, Poly 2021 and DMAH 2021, Virtual Event, August 20, 2021, Revised Selected Papers*, page 65–69. Springer-Verlag, 2021. doi: 10.1007/978-3-030-93663-1_6.

[53] L. Voicu, H. Schuldt, Y. Breitbart, and H. Schek. Flexible Data Access in a Cloud Based on Freshness Requirements. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, page 180–187. IEEE Computer Society, 2010. doi: 10.1109/CLOUD.2010.75.

[54] Y. Wei, S. Son, and J. Stankovic. Maintaining data freshness in distributed real-time databases. pages 251–260, 2004. doi: 10.1109/EMRTS.2004.1311028.

[55] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001. ISBN 9780080519562.

[56] X. Wu, X. Zhu, G. Wu, and W. Ding. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, 26(1):97–107, 2014. doi: 10.1109/TKDE.2013.109.

[57] J. Xiang, G. Li, H. Xu, and X. Du. Data Freshness Guarantee and Scheduling of Update Transactions in RTMDBS. pages 1–4, 2008. doi: 10.1109/WiCom.2008.1324.

[58] Y. Zhao and Y. Wang. Partition-based cloud data storage and processing model. volume 01, pages 218–223, 2012. doi: 10.1109/CCIS.2012.6664400.

[59] J. Zhong, R.. Yates, and E. Soljanin. Two Freshness Metrics for Local Cache Refresh. In *2018 IEEE International Symposium on Information Theory (ISIT)*, pages 1924–1928, 2018. doi: 10.1109/ISIT.2018.8437927.

[60] M. Özsu and P. Valduriez. *Principals of Distributed Database Systems*, volume 4. Springer International Publishing, 2020. ISBN 978-3-030-26252-5.

A

PolySQL Syntax - Freshness Extension

This chapter provides an extension to the existing PolySQL Syntax, necessary to work with freshness. The original PolySQL[11] Syntax will not be illustrated in this chapter.

A.1 Freshness Specification

All valid extensions for freshness-related queries must consequently begin with the keywords *WITH FRESHNESS*. They are attached as an optional leaf expression for every *SELECT* statement.

A.1.1 Absolute Timestamp

```
SELECT * FROM tableName WITH FRESHNESS TIMESTAMP '2022-07-04 06:30';
```

A.1.2 Relative Timestamp - Absolute Delay

```
SELECT * FROM tableName WITH FRESHNESS 3 SECOND ABSOLUTE;

SELECT * FROM tableName WITH FRESHNESS 3 HOUR ABSOLUTE;

SELECT * FROM tableName WITH FRESHNESS 3 MINUTE ABSOLUTE;
```

A.1.3 Relative Delay

```
SELECT * FROM tableName WITH FRESHNESS 3 SECOND RELATIVE;

SELECT * FROM tableName WITH FRESHNESS 3 HOUR RELATIVE;

SELECT * FROM tableName WITH FRESHNESS 3 MINUTE RELATIVE;
```

[11] https://polypheny.org/documentation/PolySQL/

A.1.4 Freshness Index

```
SELECT * FROM tableName WITH FRESHNESS 0.6;
```

A.2 Replication Constraints

In order to use freshness-related queries, the system needs to be consequently altered. The following sections illustrate how to transform individual Data Placements.

A.2.1 Adapting the Replication Strategy

```
ALTER TABLE tableName MODIFY PLACEMENT ON STORE storeName WITH REPLICATION
EAGER;

ALTER TABLE tableName MODIFY PLACEMENT ON STORE storeName WITH REPLICATION
LAZY;
```

A.2.2 Adapting the Placement State

```
ALTER TABLE tableName MODIFY PLACEMENT ON STORE storeName WITH STATE
REFRESHABLE;

ALTER TABLE tableName MODIFY PLACEMENT ON STORE storeName WITH STATE OUTDATED;
```

A.2.3 Refresh Operations

```
ALTER TABLE tableName REFRESH ALL PLACEMENTS;

ALTER TABLE tableName REFRESH ALL PLACEMENTS ON STORE storeName;
```

University of Basel

Faculty of Science

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)
Translation from German original

Declaration on Scientific Integrity

Title of Thesis:

Name Assesor: _____

Name Student: _____

Matriculation No.: _____

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: _____ Student: _____

Will this work be published?

☐ No

☐ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: _____

Place, Date: _____ Student: _____

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .