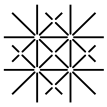


University of Basel

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

So Long, and Thanks for All the Fish. And the template.

Abstract

We summarize the feature for freshness-awareness and exemplify the implementation on the basis of the polystore system Polypheny-DB.

Table of Contents

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 towards inherently driving new technologies and scientifc advancements, becoming an industry focus-point on its own [52].
To process the ever-increasing volume and complexity of the data, companies nowadays do not only rely on their data by itself but utilize all forms of data ingestion methods to extract and connect nested knowledge out of different sources to gain economical advantages by prediciting new trends [18].
However, this also increased the need to reliably store and rapidly access such information, along 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 [43, 48]. 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 distributed data centers world wide [3]. The 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 ulitmately be achieved by replicating data throughout different regions, to provide resilient and fault tolerant architectures [13, 45].
However, in order to safely manage these large distributed volumes of data, it needs to be replicated eagerly to every particpating node to ensure a global data consistency and avoid losing data. This however impacts the accesibility of a node and reduces the possibility to be parallelly used for read access.

Therefore, cloud providers need to design their services to balance between a sufficient protection against failures and still providing adequate access times to the data [26, 31]. 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 data is not always equally important to 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 to lower 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 decission which data structure to use has a fundamental impact on the overall performance of a system [37].
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].

Despite the fact that a variety of DBMS exist which were originally created with an intention to support specific scenarios, applications are getting more complex relying on 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* [48]. Polyglot persistence is a term which refers to a practice originated from the concept of *polyglot programming*, to utilize different programming languages for different tasks following a best-fit approach [25].
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.

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 kind. 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 orginally were introduced to support heterogenous data, they

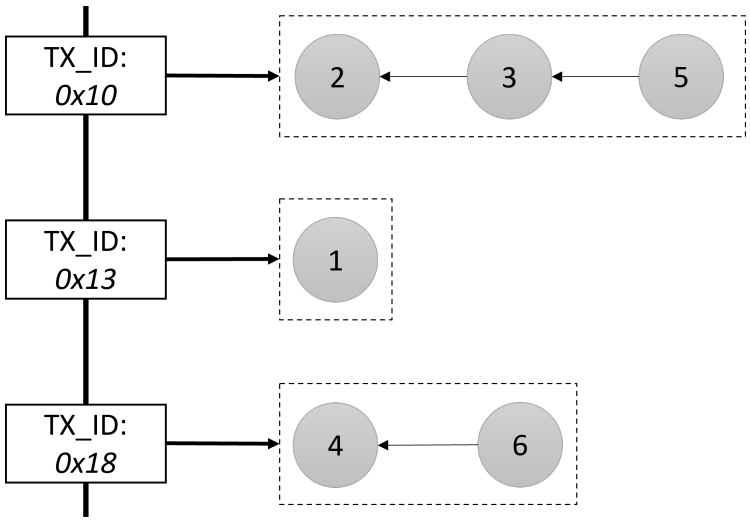


Figure 1.1: Execution-Time comparision of a single write-operation on different stores.

mostly do this by utilizing different stores at once. Since these systems are inherently distributed the data will be stored redundantly across these stores. In order for such systems to utilize the different 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. This however, limits the performance of such write-operations essentially to be bound to the slowest performing node in such a setup .

Missing: figure

As depicted in Figure 6.12, the utilized stores write the same data very differently, leading to large deviations in their execution time. This negativiely 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 again since these stores differ in terms of there 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 workload 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 obsolete 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 to efficiently use 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 a much higher throughput and increased performance in scenarios were also slightly outdated data is acceptable.

On the basis of polystore systems we could now use this scenario to nativiely leverage the benefit of a heavily write-optimized store to primarly receive any write-operation. Which then distributes the received changes asnychronously to the remaining stores, and would immediately allow using the notion of freshness to be able to utilize outdated data. This would ultimately loosen 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. 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 over the current state of research in the field of data freshness. Chapter 3 illustrates existing fundamental concepts in the context of distributed data management systems. 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 system. 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 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 and gives an outlook to future work and possible extensions.

2

Related work

This chapter aims to provide a background for the given topics as well as talking about the related and recent research activities in this field. In the context of this thesis, we consider five topics, which are necessary to establish freshness-awareness inside a Database Management System (DBMS). It is therefore separated into five sections, were each part contributes to a characteristic, needed to provide Data Freshness. This includes a definition of Data Freshness in general (→ Section 2.1), possible metrics to define cost functions which are used in the evaluation process to express and compare different freshness-levels (→ Section 2.2), how to correctly involve different versions of data items (→ Section 2.6). Furthermore, which possibilities exist to replicate data (→ Section 2.4) and propagate updates to outdated nodes in order to refresh them. Finally, this is concluded by a discussion, how users can actually specify their tolerated freshness to retireve data (→ Section 2.7).

Missing: Cite from Marco Dis [46]

Missing: this compares the work of oth systems

Missing: Check abre viation of OLTP and OLAP

Missing: Check nece sity

To ensure the overall consistency of a system, current approaches in cloud environments use well-established protocols with unified blocking behaviours. This includes Strict two phase locking (SS2PL) for correct serializability treatment as well as two-phase commit (2PC) for global atomicity in a distributed database setup. Such mechanisms are mainly utilized when updates and changes to the system are replicated eagerly to every available replica to maintain a consistent state while complying with the common ACID properties [56]. However, these directly impact and ultimately mitigate the availability and response time agreements of most service providers. Hence, different approaches were introduced which aim to relax the strict *ACID* properties towards a *BASE* assumption [38]. *BASE* in this context stands for **B**asically **A**vailable **S**oft State and **E**ventual Consistency [42]. To implement this concept, updates on data items do not need to be executed immediately on every participating node. It is sufficient to apply the change only on a few nodes and deferring the data replication to a point in time when the workload on the system is low and resources are not occupied by client requests. This form of lazily replication can drastically increase the performance while also reducing the costs of maintaining all replicas at once. However, since these client requests are not serialized anymore this could lead to users accessing outdated and stale data items which have not been updated yet. As a consequence of lazy replication the system now persists multiple versions of data objects which logically reflect the data freshness. Voicu et al. [50] suggest that since not all applications need the

most up-to-date data, one could easily exploit this side effect by keeping several replicas of a datum as different levels of freshness. This notion of or recency can then be utilized to ease the selection of correct replicas to fulfil individual client requests which even tolerate stale data as well. For cloud providers this combination of eager and lazy replication together with the resulting different stages of freshness offers a great trade off between latency and access time for freshness and enables efficient usage of available resources.

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 defintion, hence several different notions and interpretations exist, on how to characterize and measure data freshness. This was already described by Naumann et al. [32] in 1999, which further 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 the author of [40] who defined the data accuracy as "the degree of agreement between collection of data values and a source agreed to be correct". Hence, it can be considered as the precision of such data elements in respect to their most up-to-date version.

Missing: Check

This approach is now commonly used among distributed systems which compare replicas with a designated *single source of truth* or some defined primary or leader nodes in such a setup. Referring to freshness as a measure of the divergence between a replica and the current value [**?**]. Such strong application driven requirements call for a dedicated data model which represents the handling of data in a system which has freshness related requirements on arbitrary data items.

Peralta [36] also summarized freshness as an implication of how old data is, and referred this to an user's expectation on the actuality of an object. Consequently, using the last time objects were updated, to identify any accuracy measures. Peralta also distinguished between two data quality factors. The *Currency factor* which 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 the various definitions and approaches to define data freshness itself, the metrics to identify and measure the actual degree of freshness also varies. Several metrics are summarized by [16, 34, 36] and can be categorized as:

- **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**: Which accumulates the number of updates to a source since the data extraction time. For replicated systems this can be characterized as the number of pending updates that still waiting to be applied to a secondary system after it has 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 [55] define the notion of freshness by associating it to the *age-of-information* (AoI). This is defined as the timeliness represented by the timestamp of the transactions which has updated the value. Which is enriched further with the *age-of-synchronization* (AoS) that corresponds to the time when the value was actually updated.

2.3 Freshness Constraints

Along the various freshness metrics as described in 2.2, essentially three types of freshness constraints to suggest a tolerated level of freshness can be summarized [28, 56].

- **Time-bound constraints**: Are used as timestamps that accept values which have been updated by younger transactions.
- **Value-bound constraints**: This is commonly used with numbers than with text and considers the percentual deviations from the current value.
- **Drift constraints on multiple data items**: These are especially relevant for transactions which read multiple data items. Tolerates 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 [50]. 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. [41] 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 [53] and [23] 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 $|d_0 - d| < \delta$. Low values between base and replicated data reflects 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 which need to be updated. 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 [30]. This mechanism itself suggests to the user that the database system is updated eagerly while internally the updates are actually propagated lazily. Although 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 OLAP workload while still performing transactional processing at primary level as [39, 41, 53] have pointed out in their work.
Depending on the given system architecture, different refresh-strategies have been proposed on how updates are propagated towards outdated nodes.
Wei et al. [51] 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 this accessed data is not fresh enough the entire transactions will be restarted. This imposes a huge performance mitigation and wastes ressources since the transaction itself could be processed and re-queued multiple times before it might actually get executed.

Psaroudakis et al. [39] 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 fulfil these mixed workload requirements they summarized the idea of *SAP HANA*, 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 [51] 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. [35] 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 Queue (FIFO) queue to correctly serialize any updates which shall be delivered to secondary replicas.
The authors in [50] 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, [53] 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 2.4, three refresh and different propagation strategies could be identified when updates should be propagated to replicas in order to refresh them.
Despite that the authors of [39] rather focused table-level objects and not on comprehensive system replication scenarios, the up-date 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 the main area.
Röhm et al. [41] 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 afterwards. 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 [50].
In contrast to [36] which determined the goal to identify the minimum set of refresh transactions, such that it is guaranteed that a node contains sufficiently fresh data with respect to the users requirements. Therefore, any outdated node will independently pull the number of updates which are necessary to fulfil future requests along the requirements.
Wei et al. [51] 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 the consistency in a distributed setup. The authors of [51, 53] discussed data freshness together with the scheduling of update transactions from the point of view of real time applications. They ellborated 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 overtime 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 [20].
Voicu et al. [50] 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, 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 [39] 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 which have been selected to fulfil the read requests. These locks aim to provide fast response times for OLAP workload and essentially ensure that data is not refreshed aslong as it is being read by another transaction.

2.7 Freshness-aware Read Access

Since 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, 41] a router needs to utilize system state information on replicas to identify the best way to route a query.
In [50] 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 fulfil 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 are able to execute the request wthin this subtree, a refresh transaction is triggered, which will update and refresh all nodes before processing the read operation.
The authors [29] introduce a central *preference manager as a service* at the client site. This manager is able to suggest where to route queries based on given cost metrics. By comparing individual latencies for any request they aim to improve the overall performance to deliberately choose 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 on Distributed Data Management

This chapter describes concepts and general foundations, which are necessary to supplement the contents 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

Beside the utilization of the correct storage engine, the used data model and structure will also 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 [56].

Vertical Partitioning is usually applied at designe-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 [17, 33]. In order to combine and reconstruct these vertical partitions again there needs to be a logical reference, which assists to uniquely identify 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 operating 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* [26] 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 primarly introduced to support the differentiation between *Availability* and *Consistency*, it only formulates the trade-off in case of a 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 actually needed [54].
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 [30]. However, maintaining the consistency of all available replicas, results in scalability problems.
The author of [1] summarizes "three alternatives for implementing data replication". He states that these different nuances inherently result in the aformentioned trade-offs between availability and consistency described in 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 [27].
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 [23]. Although, 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 in increases the popularity for OLAP-based applications [19] 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* [42]. 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. [30]

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 [] not solely the data replication will impact the trade-offs between latency and consistency 3.2, but so does the concurrency control.

Despite the fact that there exists multiple protocols that handle concurrency control differently we will discuss two of the most common ones: *two-phase locking* (2PL) and multiversion concurrency protocol (MVCC).

3.4.1 Two-phase Locking

As the name suggest 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, that 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.
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 on 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 [] since the transaction has not been committed yet and could still be rolled back. The more strict case *strict two-phase locking*(S2PL) only releases the locks of write-operation as soon as the transaction ends. Shared-locks on the other hand are releases 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 behaviour 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 [21], 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 [19, 24].

Moreover, MVCC protocols are challenged with the decision how many version 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 any system. We will therefore discuss the contents of Chapter 2 and propose solutions how these approaches and techniques can be applied to polystore systems. Hence, this chapter is separated into several sections where each represents a necessary building block to provide the notion of data freshness.

4.1 Functional Requirements

Based on the provided discussion in Chapter 2 we have defined six fundamental functional requirements, which are necessary to 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 - *Ensure the consistency of participating stores. 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 fulfil the requirements listed above, we will define and conceptualize several possibilities in the subsequent sections tailored to be solved by polystore systems.
Except for the obvious existence of multiple replicas per data object (→ 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 (→ 4.3) and further provide users with a possibility, to formulate an acceptable level of freshness (→ 4.7). Based on these fundamentals we need to consider how update transactions can be decoupled and defer the refresh of specific replicas (→ 4.4), all while ensuring the consistency of the entire system (→ 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 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 polystores 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 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. 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 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 is 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. Which 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 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 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 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 enacts a lower bound of 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 although 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. Although 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 therefore also need to specify the accepted level of freshness based on the divergence from its eagerly replicated counterpart and therefore 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 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 make an assumption on 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 [41]. This ratio can be evaluated and consequently generated based on the number of modifications the primary and secondary copy 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 *JOIN*ed. We can sum the joint number of modifications and compare it against the current number of update transactions to give a joint accumulation.

Since it also might be desirable to specify a freshness index but to consider a time deviation as described in [28, 50] 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 filtering 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 don't 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 fulfil such a request (see 2.7). This enables us to use a polystore to fulfil 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 to 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 fulfil 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, to two subsequent transactions. While the eagerly applied modification is executed synchronously it is bound by the slowest performing node in a setup. That is why 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 and 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 its 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 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. That is why an on-demand approach is necessary to refresh outdated nodes at once and bring them up-to-date. Applying such an approach, the queue has to remove all pending updates for that specific replica, to avoid that updates are not applied twice.

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 polystores 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 therefore again 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 obviously 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 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 prima ry. This assumes that if an outdated replica has been read and is above the centrally configured threshold, that no update propagation will be automatically scheduled. This will avoid 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 First-In-First-Out (FIFO) queue. Since this is only a preliminary step we will conveniently call this capture-collection: *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 deligating 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 transac-

tion. 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. Although 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 by any freshness-related queries. Hence, we have to refrain from providing this replica as a possible candidate in the retireval 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 the 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 indvidual 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 fulfil 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 2.4 there exist several techniques how outdated nodes can be updated lazily 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 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 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 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

As already mentioned, 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 briefly described in section 4.2 the usage of lazy replication is already enough 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 transactions types to allow a deferred refresh of objects. Update transactions in this sense are transactions that contain at least one write-operation.

- **Update transaction**: Consequently are 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**: 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 alreBacause ady have a pre-defined execution plan with a pre-determined set of operations that is going to be executed on outdated replicas.

Although, logically being used differently they are technically executed with the same capabilities and only really differentiate 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. 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 which part of the data resides, we can redirect all queries to fulfil 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 predefiend 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 or break 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 4.6 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 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 artifical 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 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 towards 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 [50].
For this, we always require that there is at least one up-to-date replica that contains all neccesssary information or consequently as many up-to-date replicas that they 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 even 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 this requirements is the usage of referential integrity. But despite that a polystore 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 sucessfully serving the query. This is then however done omitting the advanatages of freshness-awarenss 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 you can only read newer and never data older than you have already obtained. This also means that if you needed 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 ellaborated 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. Afterwards 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 for freshness within Polypheny-DB.

5.1 Polypheny-DB

The implementation is based on the polystore system Polypheny-DB[1]. In this chapter we briefly describe and illustrate a simplified version of Polypheny-DBs current architecture. As well as some fundamenetal components that will be throughout this chapter
This extends the foundations laid out in Chapter 3 and sets them in context of the existing system model.

Polypheny-DB is an Open-Source project[2] developed by the *Database and Information Systems* (DBIS) group of the University of Basel. It is a self-adaptive polystore that provides cost- and workload aware access to heterogeneous data[47].
Compared to other systems like *C-Store*[44] or *SAP HANA* [22], Polypheny-DB does not provide its own set of different storage engines to support different workload demands.

[1] https://github.com/polypheny/Polypheny-DB
[2] https://polypheny.org/

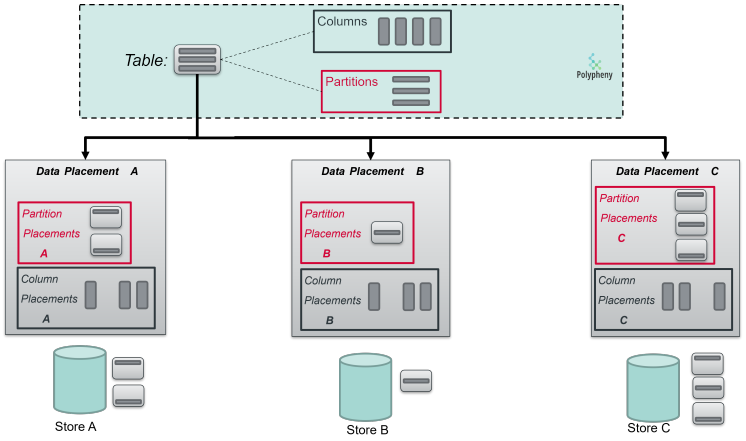


Figure 5.1: Polypheny-DB's entity representaiton via placements. Mapping a logical entity to the actual physical stores.

Instead, it acts as a higher-order DBMS which provides a single-point of entry to a variety of possible databases like *MongoDB*[3], *Neo4j*[4], *PostgreSQL*[5] and *MonetDB*[6]. 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 desigend 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 leverages Polypheny-DB to a data orchestration platform towards an actual PolyDBMS [49].

Missing: polypheny support multi-model databsaes for relatio document, graph in memroy ...

5.1.1 Placements

Placements are considered to be Polyphenys virtual representation of physical entities. Since Polypheny provides a multi model approach, underlying stores can range from relational to document until graph engines. Entities are therefore a genereal definition encompassing tables, collection or graphs, respectiviely 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.

Missing: remove rep cation strategy from image

Data Placements A Data Placement is essentially a virtual representation of the physical entity residing on a given store. A store in Polypheny is an underlying physical data

[3] https://www.mongodb.com/
[4] https://neo4j.com/
[5] https://www.postgresql.org/
[6] https://www.monetdb.org/

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 Placment 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 partitoning 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. Additioanlly, Polypheny suppports the most common partition algorithms like *HASH*, *RANGE*- and *LIST*-Partitioning. It allows to selectiviely query independent placements respectiviely distinct underlying physical stores, to distribute incoming workload evenly within the cluster. Together with Column Placements they provide great flexbility to adapt and customize any system, to fit various requiremnts.

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 fulfil a given request or to provide a given result.
The Routing Process can roughly be distinguished in 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 cachable object to simplify further routing steps. 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 build 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 entitites, allowing for a large set of possible combinaitons, this phase can be cached. This enables pre-delivering already geenrated plans for similar queries to be used by the last phase to save processing time and reduce overhead [46].

Since every query has to go through the abstraction layer to guarantee correctness and consistency, Polypheny-DB can consult the systems 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 Polyphenys current architecture all incoming queries have to be delivered through this central polylayer, acting as a central 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 respectiviely 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 ultiamtely commited by all underlying stores, which introduces a bottleneck for this layer, that is inherently dependend on the slowest performing store.

Missing: Maybe rename?

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. Which again shall help to reduce the overall latency of the system by allowing mixed work-

load to exist in parallel.

Missing: In correspo
dence to the section
update propagtion in
concept we focussed
implementing the CI
approach (→ 5.2.6) a
well as the on-demar
approach using pri-
mary Snapshot Copy
(→ 5.2.7))

5.2.1 Placement Versioning

As we have established in section 4.2, the existence of multiple data replicas are 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 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. However, as stated in 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 severall stores. Furthermore, this would also consequently imply the utilization of MVCC. But aforementioned, 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 reponse 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 in freshness related queries.

Missing: Besserer
Übergang

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 to create multiple data placements for an individual entity. The introduced data placements described above can therefore be considered as an individual replica or version for 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 autoamtically 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 to base our freshness awareness on.

5.2.2 Replication Strategy

In order to reduce the update time per write-operation and increase the performance of OLTP workload, 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 placments on how they are going to receive updates. This is defined as the *Replication Strategy* $\Gamma \in \{EAGER, LAZY\}$. $EAGER$ means that DML operations 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 in the beginning of this chapter, partition placements inherit their information from its corresponding data placement. Although they receive there 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 enitrely 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.
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 Stateemnt 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 column 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 conssitency and integrity of the data (see 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 Is automatically set within a partition placement, when the parent placement is configured to receive updates eagerly. This cannot be changed by any user. It does also not refer to the current state of any data object, meaning that an lazily updated placement can become up-to-date overtime. Allthough 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 is specifically marked, 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 surpressing the automatic update replication on this node. Additionally this can be set autoamtically 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 an *REFRESHABLE* state.

The distinction between these cases is necessary, to allow treating partition placements on a given store differentely. 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 specifcation of this state is still targeted to an entire data placement (5.2). Since the internal partitioning should be rather user agnostic one should only be able to specifiy this per data placement. As with the replication strategy, the changes are then propagated downwards to all linked partiton placements.

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 OUTDAZED 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.

```
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.

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 DML-operation, the router can identify for a given entitiy 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, as well as the current statement and 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 its statement id to a preliminary *capture-queue* within the *Change Data Collector*. As visualized in 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.

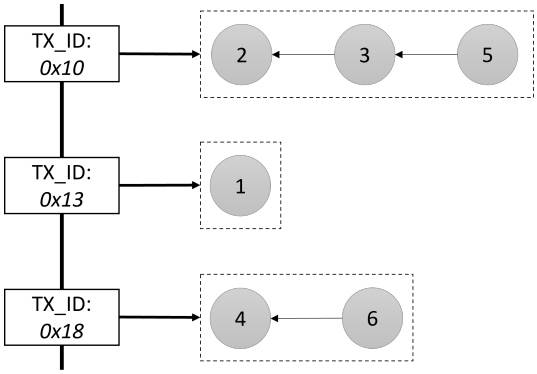


Figure 5.2: Capture Queue, containing pending transactions as well as an ordered list of executed statements containing the captured data.

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 commited.

5.2.5 Lazy Replication Engine

A *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 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 orginal one, that has already been executed on the primary placements. Therefore it keeps information on the orginal transaction,

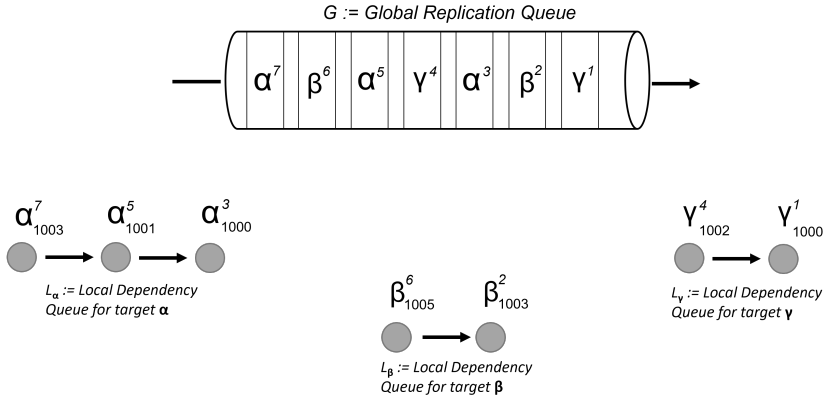


Figure 5.3: Association between global replication queue and local dependency queues.

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 DML-operation. This data is kept as long as there are replication objects depending on it for executing their replication.

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 identifiying 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 it 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 left of the dependency graph. Resulting 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 proccessed 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 form 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 Dependecy Queue* always will ensure the correct execution order a concurrent processing of the replicaitons is possible.

Data Replicator Is implemented as the actual execution engine which 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 advanatage that we can now decide based on the targets 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 could have been altered since capturing, it could now have a different set of columns.

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. These information will be updated at 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 numer 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 a high flexibility.
Furthermore, it does not only replicate the captured modififcations 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 centrally stored and is only removed when all dependent replications have been executed, a replication event that coninuosuly keeps failing will uneccessarly occupy worker threads and waste storage resources to preserve the data. Therefore a centrally configured *Fail Count Threshold* is introduced. Everytime a replication fails, the responsbile worker thread will increase the *fail counter* of a given replicaiton object. If it reaches the threshold the replication will be removed from the queue entirely. Furthermore the traget will be removed from the engine by deleting all remaining local replications for this target as well. Consequently this particular partition placement will be suspeneded 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 replciation distribution. In constrast to labeleing data placements as outdated, and removing all pending replications for ever, 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 funcitonality.

5.2.6 Automatic Lazy Replication Algorithm

The goal of the entire lazy replication approach is providing 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 DML-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 conseqeuntly have to be updated lazily. That en-

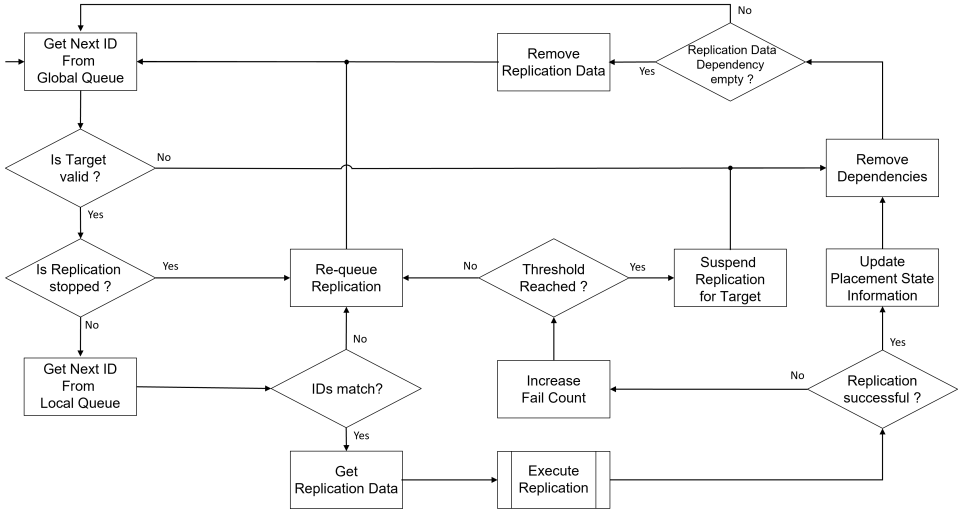


Figure 5.4: Lifecycle of a *Replication Worker* processing pending replications from central queues.

abels the entire transaction to *capture change data*. This will directly result in transforming the write-operation of the current statement into a set of basis operations, that are already evaluated and can therefore be immediaetely 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 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, neccessary to execute the statement. These parameter values are added to the *Change Data Capture Object* as well. Accompanied with 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 5.2).
As soon as the transaction has finished, this object is further processed. If the transaction was aborted and rolledback, 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 succesfully commited, 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. Afterwards 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. Additonally, 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 prelimiary 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 plaecments, and ensures that they uniformly progress towards a given state as its up-to-date counterpart has.

Since this is all still done before the transaction has publicly finished, further operations are blocked until we have asured 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 went wrong during queueing, we can still relabel those placments 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 queing process has finsihed and the primary update transaction has returend, removing all locks, the *Replication Workers* will eventually process the queued events. 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 moving 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 automatically cleanses the queue from all remaining replications associated with this placement. Assuming that the currently proccessed event is indeed the next replication in line it will prepare the execution and start a new refresh transaction. The data replicator will now analyze the replication event, and ultiamtely 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. Additonally 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 savely removed to reduce the data footprint of the system again and freeing up resources. 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 threhold, 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 behaviour 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. However, 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 onto 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 autoamtically switches the replication state of this placement to REFRESHABLE, starting to activiely 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 placmenets, 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, it 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

Since we are in a distributed setup, we always need to ensure that we do not lose any data, while transforming the indidivdual placements. This already starts by defining the replicaiton strategy. Although Polypheny-DB allows to customly distribute an entity across several stores, we have to enforce that no information is lost. This means that since we can arbitraly place any combination of columns and partitions of a given entity of any store, we need to make sure that at 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 again have to ensure that at least the eagerly replicated placements are sufficently configured such that each column is available for all partitions at least once. The remaining secondary placements however can again be arbitrarly 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 currently are 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 behaviour 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 propagting changes towards this placement again, we will lose data the data in between thos versions. In this scenario the system first will need to make sure that the placements on this store are all uptodate. If this is not the case the operation will fail. This can be 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 utilzed queues within the engine are still only stored within main memory. Although this increases 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 behaviour. Since we always know which placments are updates 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 inforamtion on all placements that need to be updated lazily. It can automatically mark them as INFINITELY OUTDATED, and therefore suspending any more replications. Users can either choose to update the system manually or refreshing is on read as proposed in 4.4.1. 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 defferal of transactions prepared, we have already established a foundation to use those possibly outdated versions for freshness-awareness.
This section therefore focusses on all aspects to establish the core aspects of freshness within Polypheny-DB, to allow users the specification of freshness and efificiently using those outdated placements to distribute the workload across the system. The actual freshness evaluation can therefore be separated into roughly 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 primarly used to filter placement candidates based on a defined freshness based on their *Update Metadata*. We have summarized these possibilities and established three different evaluation types, that can be used to specify an accepted 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 there commit timestamp is newer than the specfied 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 accetable delay for the desired freshness. This will intuitively substract 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. Again as described in 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 Naturally it 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 accumultaed 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 that 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 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. 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.

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

Listing 5.4: Generalized Freshness Specification

The statement depicted in 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 analzyed 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 4.6, this will transform the transaction to read-only mode, prohibiting the execution of any more DML 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 conventonal 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. Beside 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.

Although, not neccessary 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 2PL, 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 interefere 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

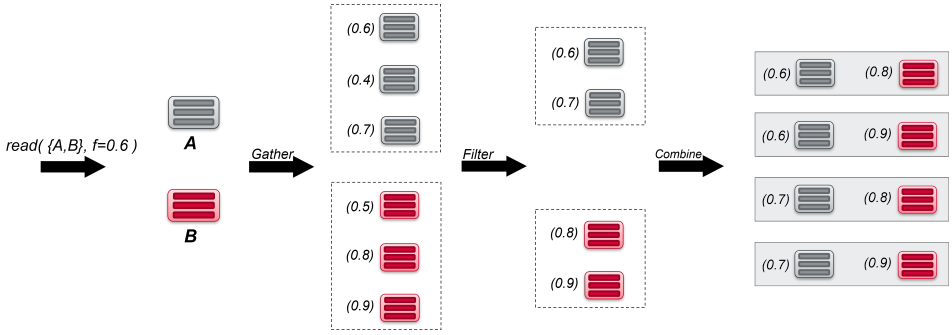


Figure 5.5: Subsequent Filter Operations for an abstracted freshness read operation, generating several possible execution plans based on a given freshness-index.

the possibility for any new upcoming freshness transactions to add itself to the existing shared lock. Instead they are treated and added to the dependency graph of the refresh operation an 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.

Missing: implement

While the previous approaches simply tried to provide any possible combination and construct a result out of different independent versions, the last charcteristic goes a step further and allows to specify a referential integrity enforcement. As described in section 4.7 this approach aims to enforce the referential intergity 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 is 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 even has several versions. Therfore 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 fulfil this request, we always have the possibility to fallback to the primary nodes, which are guaranteed to support referential integrity. Allthough, this again will again block updates on the primary placements, it fulfils the constraints as it would have, when executing the same query without specifying a tolerated level of freshness.

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 transform 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. 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 ist 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 is 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. Aforementioned 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. These information can be consequently used to compare their state against the corresponding primary placement with the same partition. While *TIMESTAMP* and *DELAY* can be used directly to assess the cadidates, 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 placments are only kept among the candidates if they fullfil 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 acccumulated freshness when combining the placements (see 5.3.5). Since our query can require multiple partitions at once, we could now combine a very fresh placement with a placement that is by itself considered to be too outdated. While the individual placements might not fulfil the placement, their combination migth 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 intial pre-mapping of partitions we can resepctiviely 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 appoach, targeting only primary placments as if freshness has never been specified.
Although 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 allowing 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. Additioanlly 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 to utilize 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 fulfil the request. While it is always save 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 expenssive 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 indivudally specified attributes. These could be allowing dirty reads, ergo allow 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 specifiifcation, we can validate if we have already cached a similiar 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 again provide access to outdated data .
This ultiamtely enables freshness-awarenss within Polypheny-DB to evaluate and utilize the freshness specififcation, to improve the inherently distributed system architecture to decouple transactions and efficiently distribute workload to utilize outdates replicas and consequently increase the performance of the entire system.

6
Evaluation

This chapter is separated into two sections. The first section aims to validate and verify the implementation in terms of their correct execution. It focuses on the core contributions, such as the lazy replicaiton 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 possibile failures that can occur during execution as well as suitable failure handling scenarios.
The second part of this chapter focuses on benchmarking the implementation on the basis of the described motivational scenario itself. Since one of the main goals was to relax the consistency and allow This is followed by comparing the performance of several cases to identify how the system will behave in certain situations. Further it will compare several executions to determine how the impleemntation affect the providede requirements.
Furthermore the performance of the individual functionalities is benchmarked and compared against slightly adjusted variations to give an comprehensive overview of the impact.

6.1 Goal

The evaluation has two goals the correctness as well as the impact of data freshness onto different kinds of workloads.
Verify and validate the correctness as well as the completeness of the implementation based on several characteristics. These include the correct execution of lazy replication, the possibility to refresh statements on demand.
Impact of the replication engine on the underlying performance, if freshness indeed increases the overall parallel writes on the system. Or if it is just marginally lower than before. Also compare this to the overall introduced overhead. And if the change was wort it
Compare underlying stores to assess the deviations of the polystore in total.

6.2 Correctness

The correctness of the introduced solution mainly focuses on two parts. For one the replication behaviour, to verify if each lazy replication is carried out correctly, and if not verify

that reasonable counter measures are in place 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 on a high level the metadata (i.e. if the number of modifications and the commit timestamp after the data replication are equal on primary and secondary node) of two replicas.
The second part of the validation process focuses on the retrieval of outdated nodes. Although we always have a fall back to the primary placements as described in section **??**, we still want to avoid excessive locking to parallelize requests to ultimately speed up the average response time.

6.3 Benchmarks

6.3.1 Evaluation Environment

If not explititly stated otherwise, the benchmarks will be executed using two underlying stores.
Define that we have HSQLDB fast and PSQL slow because on docker.
First show single Performance on the Store to see which one is the slower one of these stores in our setup. This will also show which stores logically bound the primary transaction time.
PSQL on Docker. Since Docker Containers use a limited shared set of resources this store immitating a slower performinh node in oru setup.
Define Workload Environement DQL DML Mixed Worklaod if not stated otherwise is 60% read-operations and 40% write-operations

6.3.2 Evaluation Procedure

The following steps outline the procedure for benchmarking data freshness within Polypheny-DB.
progressiviely build on top of each other, test after test.

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, addtional introduced overhead 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 system.
Hence we used Chronos a benchmarking tool.... to measure the overhead. For this evaluation the introduced implementation is compared against the current state of Polypheny-DB.

As depicted in Figure 6.1, the results indeed show that the implementation introduced a

Also checks for freshness specification if t can for one be even a plied to the system , due to ongoing constraints or if it is eve posssible to specify a freshness index ¿1.0

Missing: **??**

Missing: As already suggested within con trianst we have to en sure that data will n be lost

Missing: How to ensure that cached fres ness objects are not falesely executed and secertly violating the specified freshness (I rechecked)

Explain why it is nec essary to verify the s lution with different combinations

Elaborate and thoroughly explain why t environment was cho and how the test was executed for the sake rerpoducibility

Check how fast the replication is compar to the primary execu tion. Benchmark on two equal stores and measure the time

Execute benchmarks multimodal dbs. as

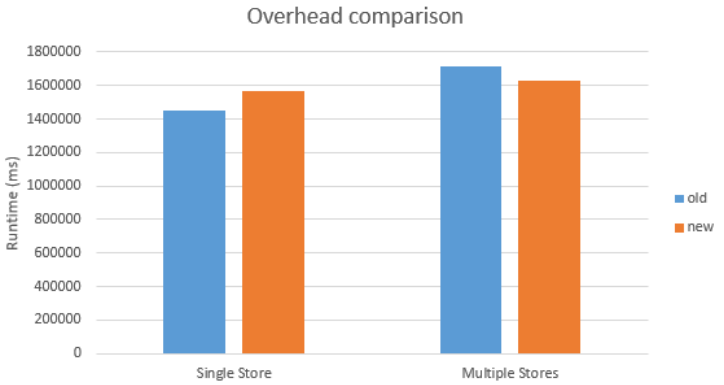


Figure 6.1: Overhead comparison of the entire implemation.

little overhead for single store operations. However with multiple stores the actual average execution time was indeed slightly reduced. Although, single store executions should not be entirely negelcted they do not form the main pursuit of polystore systems, which are more suitably visualized by the multistore runtime.

Missing: Contain overhead as much as possible otherwise the freshness eveluation will negativiely impa regular operations

6.3.2.2 Locking

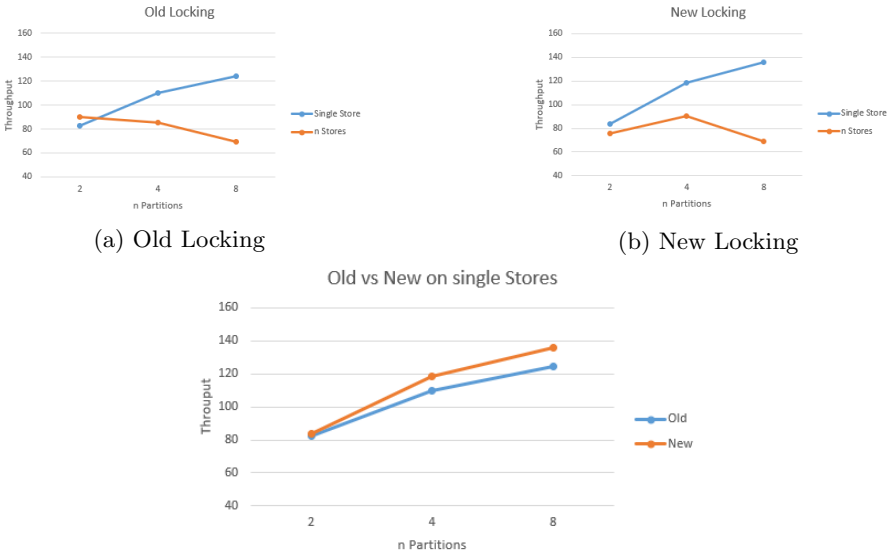
As described in Section 5.2.2, one of the prerequisites to establish multiple refresh strategies and hence lazy replication, was the refactoring of the locking mechanism. Although, not completely reworked, the locking-module of Polyphenys SS2PL poses as a core component of the system. It therefore impacts correct serializability treatement and is and inherent driver of the 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 towards a partition-wise locking we will validate the impact on the basis of a single table using YCSB.

Missing: cite

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 on the locking and therefore the throughput.
To get a general overview of the impact, the benchmark was executed using a mixed workload.

As visualized in Figure 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 behaviour is essentially caused by the 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 6.2c. We can observe that the new locking mechanism indeed proves to be slightly better in terms



(a) Old Locking

(b) New Locking

(c) Old vs. New Locking

Figure 6.2: A figure with three subfigures

of the possible throughput. Furthermore it shows that again with a growing number of stores, the gap between the old and new locking extends even more, validating the benefits of the new locking mechanisms.

6.3.2.3 Baseline Identification

Missing: Single PSQ vs HSQL

Since the Lazy Replication algorithm is not only fundamental to generate multiple versions to be used within freshness-awareness but also a core functionality 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 perfromance of the system. Since the replication solely focuses on replicating captured changes, the next benchmarks will be consequently executed using only DML-operations without any reads.

Missing: ref section

As stated in the evaluation environement , these benchmarks will be mainly executed with HSQLDB and PostgreSQL running within a virtualized container environment. To have a general base line for comparison Figure 6.3 presents a single store execution, comparing theses two stores against each other. As motivated in the beginning, it is crucil for a system to utilize the key benefits of each store. For our scenario this is important to determine which store configuration is more suitable to be used as an eagerly replicated priamry placement, due to its lower latency and better reponse time.
This illustreated comparison clearly shows that Due to its limited resources the PostgreSQL store cannot directly compete with this HSQLDB configuration. This provides us with the intuitive decission to use HSQLDB for the priamry transactions.

6.3.2.4 Lazy Replication

Missing: TERMINA vs TERMINAL 50

As previously stated, the replication strategies will impact the processing capability of the

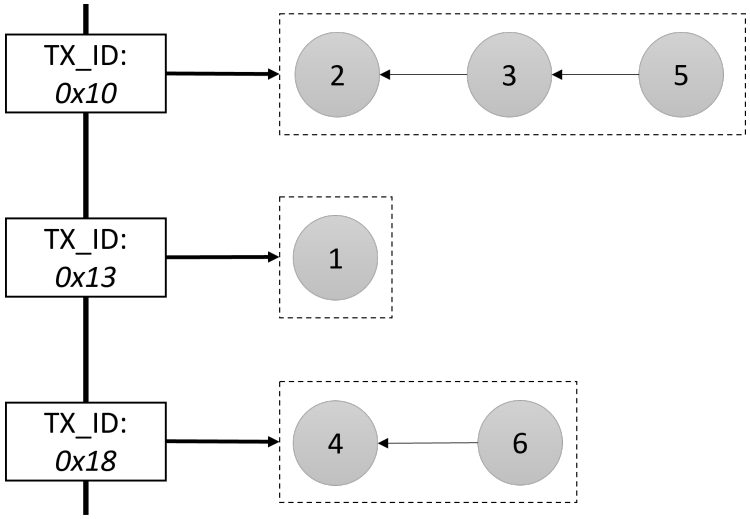
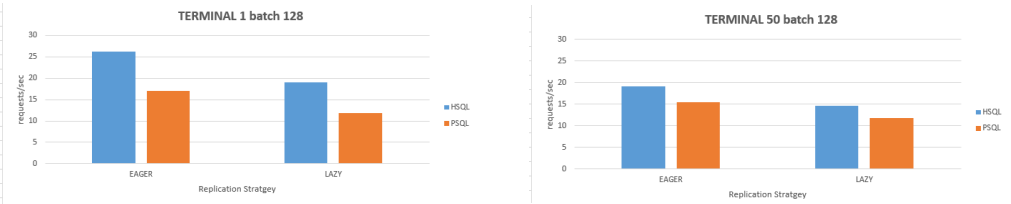


Figure 6.3: Single DML PostgreSQL vs. Single HSQLDB

system immensly. 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. Therfore we want to compare how each store handles the replication differently. 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 behaviour 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. Which is given as the number of modifications that can be applied to the system per second. As before HSQLDB obviously achieves better results then PostgreSQL. However what is surprising is that disregarding the store, and not, apparent when only considering 6.4b, the eagerly replicated configuration performs much better in all tests. Considering that he collection of changes within a llazy setup, indeed imposes additional costs on the processing time, such deviations are expected.



(a) One process (b) 50 processes

Figure 6.4: Concurrency Comparison

Admittingly an enitity that is composed of only similiar or equal stores, will not be beneficial for a polystore system, to allow different workloads. Therefore the follwoing benchmarks will concentrate on a mixed setup with interleaved stores. Furthermore these tests will be

Figure 6.5: DML only - Avg. Latency comparsion PostgreSQL vs. HSQLDB. With switching Roles

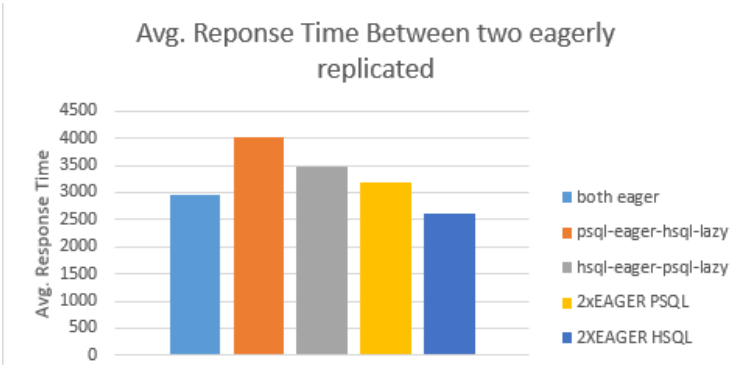


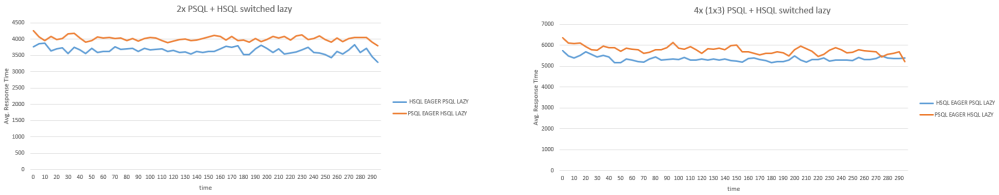
Figure 6.6: DML onlyAvg. Response Time Comparison of 2 store sizes and swithcing Roles

executed with 50 parallel clients to repoduce a conventional environment.
Now focussing on a mixed setup of two stores containing PostgreSQL as well as HSQLDB for one entity. Nativiely fo a polystore environment we want to identify which setup will produce better results, hence is suitable for which situation. Consequently we want to observe how the order of the stores imapcts 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 6.5 shows that regarding the lazy approaches, we again see the common behaviour 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 pefrom the best while the PostgreSQL variation posing a eager perfroms the worst. This not only allows to compare the different setups but again aids us to choose suitable combinations for our designated tasks.

Missing: PSQL vs HSQL DML only

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

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.



(a) Comparison of 2 Stores HSQLDB and PostgreSQL. Switching Roles.

(b) Comparison of 4 Stores HSQLDB and PostgreSQL. Switching Roles.

Figure 6.7: Comparison of different store sizes and swithcing Roles

As summarized and visualized more densely in Figure 6.8, its rather counter intuitive that the configurations in 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. However, as described before, the primary transaction is responsible for capturing, as well as queing 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 priamry transaction, the observed deviations are reasonable.

Based on this observation we specifically wanted to compare how a growth in stores will impact this deviation. To have a more uniform result this test will be executed with two HSQLDB stores, to have a simple foundation to compare each deviation . Without a secind store chararcteristic that could interfere with the final result.
In this evaluation 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.
HSQL only Compare execution time with different number of stores 1 Eager n Lazy (Because for Lazy there are more replications to put into the queue per pending operation) Check the average repsonse Time

6.3.2.5 Queue Replication

Mixed 60-40
Shows the impact the queu operations have on the overall performance
As ellaborated and suggested multiple times before, the deviation between eagerly and lazily replicated deviates quite a bit. Therefore we want to identify per operation what actuallay influences the lower resposne time.
Therefore show how this deviates in General. What happens if we disable the replication queue or teh capture in geenral. To immitate an on-demand processing where entities and placements are purposely kept at a given point
We see exactly the delay starting after 1 5se minute because afterwards the first placement

Figure 6.8: Avg. Response Time Comparison of different store sizes and swithcing Roles

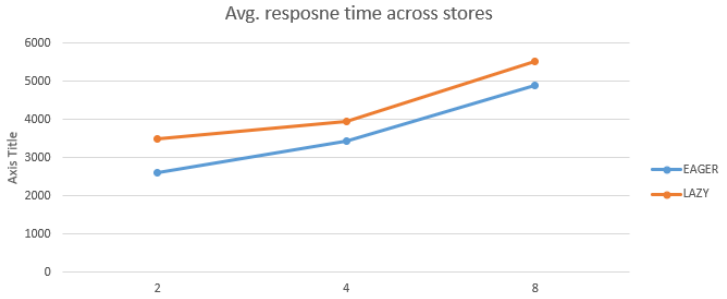


Figure 6.9: DML only multiple STore comparison Response Time

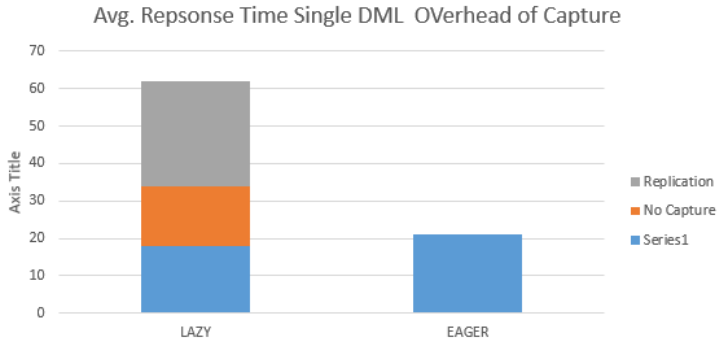


Figure 6.10: Execution-Time comparision of a decompossed single write-operation with and without data active data capture

could fulfil the freshness request and had to fallback to the primary who have might been locked This graph exactly corresponds to the graph depicted in 6.10 Mixed 60-40

6.3.2.6 Compare Total Replica Convergence Time DML only

BOTH eager – PSQL EAGER HSQL LAZY – HSQL EAGER PSQL –LAZY In terms of their execution and replication time which one finishes faster Also does the numbe rof stores have an impact Mixed Worklaod?

Average Repsonse Time accumulated with the Extension of teh Replicaiton Queue as well

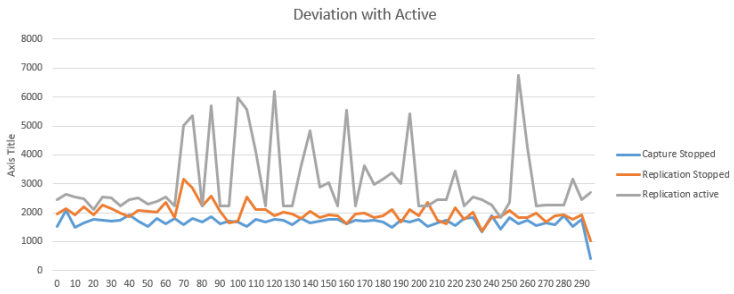


Figure 6.11: Mixed Workload 60-40 with 100% Freshness with relative read 1 minute

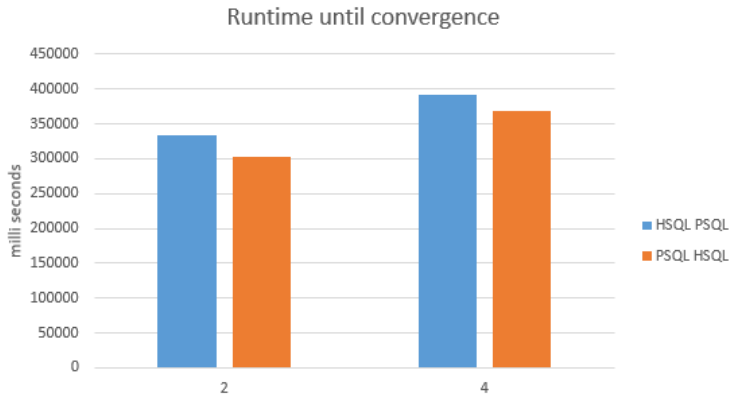
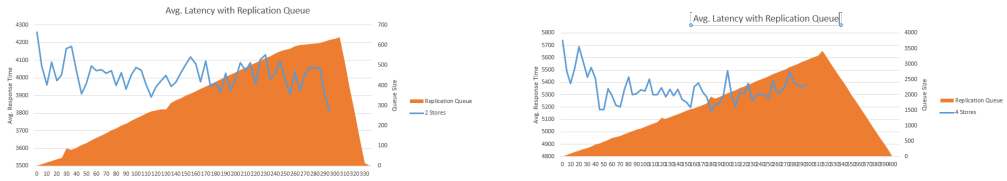


Figure 6.12: Convergence Time of PostgreSQL and HSQLDB with n-number of stores.

as the shrinking to mimic the convergence time.



(a) 2 Stores

(b) 4 Stores

Figure 6.13: Execution Time and Replication Convergence

Reponse Time DQL in General

DQL only

Even with mixed workload without freshness as illustrated in Figure 6.15, it is obvious we see that the store consteallation does have an impact on the overall latency. Whereas no freshness queries will always target the primary playement we see again that the store with HSQLDB as a priamry node is slightly faster than the PostgreSQL Eager variation. This again proves the point that the Store selection will have an impact. However know compared to a query with freshness, we essentially see very flipped roles. . This is essentially caused by the target of the select statetemntes. Since we are ratehr in a read-heavy environemnt the select to have quite the impact on the final result. in contrast **??**in that case targeting

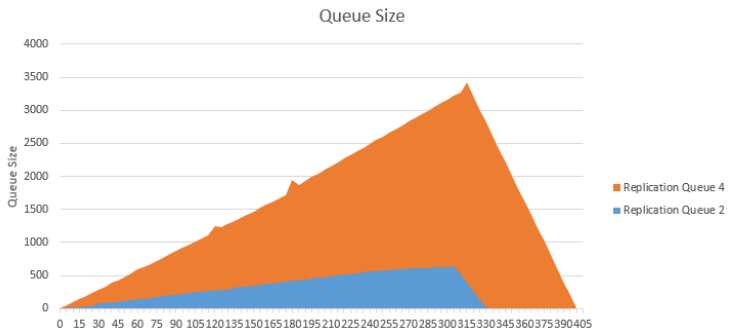


Figure 6.14: Replication Queue Convergence

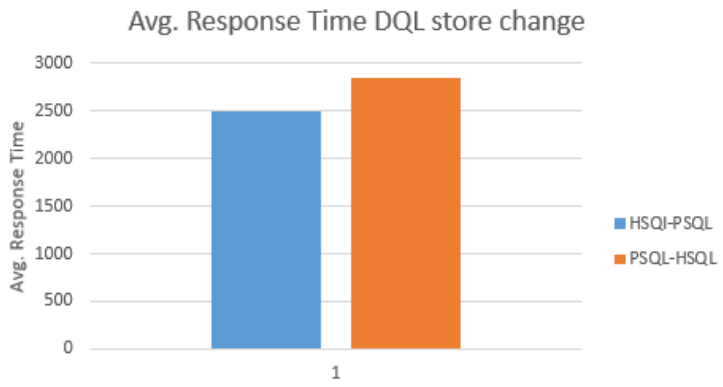


Figure 6.15: Mixed Workload 60-40 No freshness

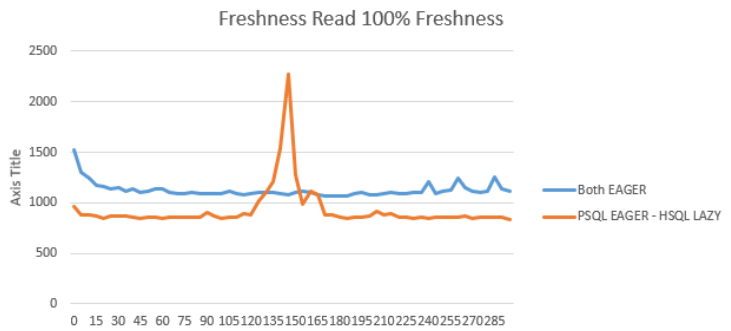


Figure 6.16: DQL 100% Freshness

the secondary node. Where again the lazy replicated HSQL Node is better

No Freshness

100 % Freshness

DQL 50% Freshness

Missing: Compare w both EAGER

Missing: Introduce 0 Freshness

6.3.2.7 Freshness Evaluation Type Filter

Show how each filter impacts the store differently, but all in all roughly the same

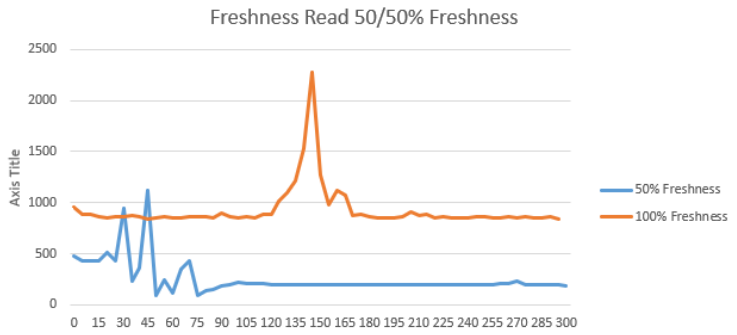


Figure 6.17: DQL 50% Freshness

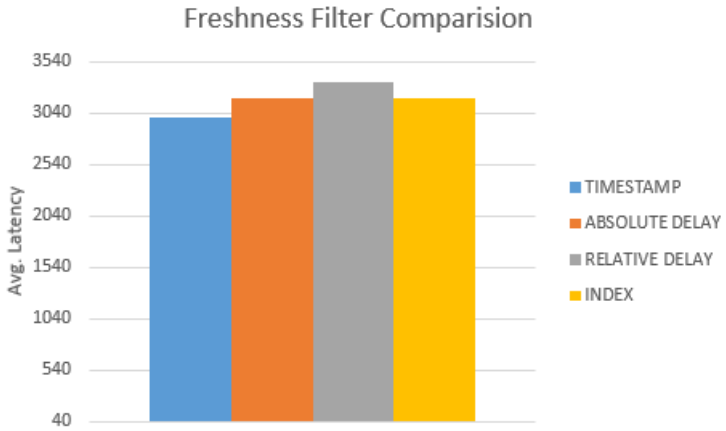


Figure 6.18: Avg. Response Time comparison of the available Evaluation Type Filter Functionality

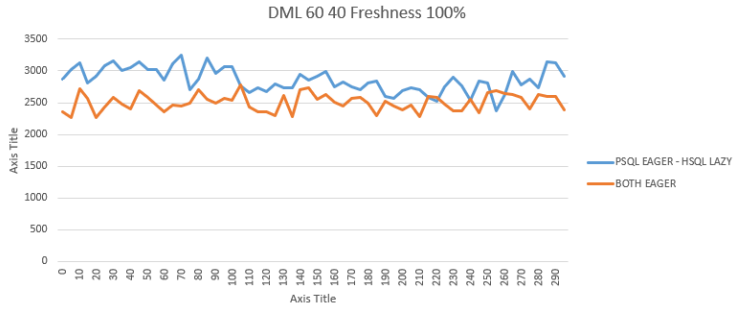


Figure 6.19: 60 R 40 W 100% Freshness

6.3.2.8 Mixed Worklaod

Show that the workload has a direct impact on the performance

6.3.3 Results

The result generally shows

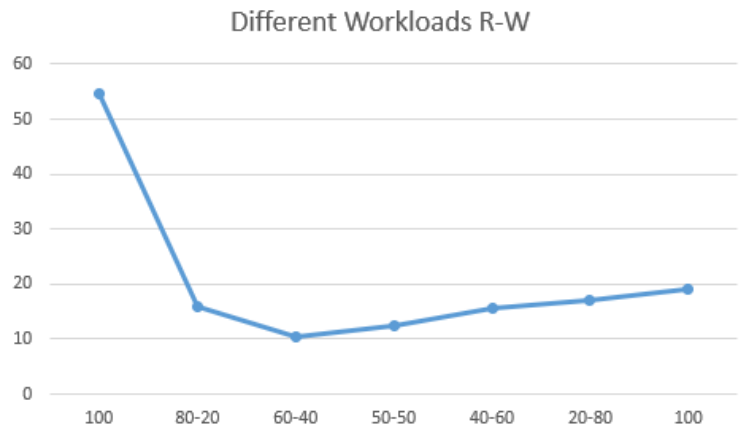

Different Workloads R-W
60
50
40
30
20
10
0
100
80-20
60-40
50-50
40-60
20-80
100

Figure 6.20

7
Conclusion

With this implementation Polypheny-DB now provides functionalities to adjust itself to the concepts revolving around *CAP* and *PACELC* described in 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.

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 introduced 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.

This immediately enables us to use possibly outdated nodes containing stale data to be used during retrieval to support analytical queries even in the presence of transactional load. This does not only improve parallel processing but also allows the efficient usage of all available stores.

With this implementation we have introduced a fault tolerant replication algorithm which can be used to lazily replicate changes to outdated replicas, while ensuring the correct consistency of each placemnts by enforcing the natural execution order. All while autoamtically rescheduling failed replications and removing left over replications from suspended or removed placments, hence cleansing the environemnt during runtime.

The results presented in section **??** show that allthough freshness can greatly improve the performance of already the differentiation between eager and lazy replication, helped to drastically reduce the average repsonse time of the system. Furthermore it now enables Polypheny-DB to decide per entity how the trade-off betwwen latency and consistency should look like, in accordance with the CAP theorem.

When partial replication is used, several of the underlying stores may qualify for the execution of these queries. In order to avoid that single stores are overloaded, query processing and optimization can effectively consider version selection and load balance the access among relevant replicas.

Although this work has shown in **??** that is greatly improves the throughput of this inherently distributed system, the usage should still be considered with care. Despite that we

established certain counter measures
At the end this work introduced several nuances of freshness to support varying use cases and requirements. Albeit not being able to support Serializable Snapshot Isolation, the implementation still offers
Since we are certain, that we do not have infinitely available versions as in conventional multi-version database systems. Even the freshness specification without any determined tolerated level, promises a certain level of freshness by design.
Of course there are obvious choices to improve such as teh queue persistency. While enabling a fast access when storing replicaiton in memeory it is prone to failures and outages, losing relevant data to be replicated. In order to remove the workarounds during startup, teh replication queues as well as the data have to be persisted. Despite slower processing times for priamry update transactions, we would increase the stability and the receoverability of the system immensely At the end this again is a trade-off between availability and consistency, dependening on indidivdual requirements. This is however is an obvious extension for this work and introduces a foundation for future work.

7.1 Outlook

7.1.1 Tuneable 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. Hence, an extension to the described model could easily allow to adjust the required consistency as needed. This could be either done by the mentioned usage of policies, described in section 7.2.1 or with.
Instead of labeling fixed data placements to receive updates eagerly, we could allow a more flexible approach that ts is sufficient if already placement shall receive the update, disregarding its role. The predefined replication state can be therefore omitted. Such approaches can then be easily combined with tuneable consistency to allow self adjusting data placements adapting to individual use cases.

7.1.2 Locking

Reduce locking to a physical partition level (partition placement)

7.2 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. Although this introduces a high degree of flexibility, it still might be desirable to define certain policies that entire schemas or even databases automatically receive a lazy replication, while still ensuring the overall placement constraints.
This concept could be intended even further by applying it to a distributed setup of Polypheny,

Missing: Allthough not all entities have support freshness, bu since we allowing a f back to the respedtiv priamry placements, users can act agnosti of the underlying arc tecture and specify t freshness eitehr way. Since we can always fallback and route th query towards the pr amry placements. Th enables user agnostic access and further al lows to configure the freshness values with an application. That would then autoamti cally evaluate the fre ness once it actually has different versions data.

Missing: In regards t CDC, if we observe that the number of pending update oper tions exceed a certai threshold for exam- ple 50% of the total number of modifica- tions of the master v can directly remove pending updates and execute a primary sn shot copy because th is faster then reexecu ing the operation ag

Missing: Explain po- tential optimization steps that we can an alyze the queue and

that replicates data autonomously to certain regions based on the given This extension could leverage the introduced freshness-awareness to consider off-site locations for even more parallel workload.

7.2.1 Policies

According to the idea, to generally relax consistency or allow a fine grained way of letting data owners decide what kind of consistency shall be desired for their object. Since freshness can be considered a trade-off between availability and consistency it is only fair to let users decide which level of consistency to enforce and how the freshness should be handled. We therefore propose the notion of policies to guide the system. Policies are essentially intentions and desired states how the system should behave in various situations. The system can apply them when manual or automatic system maintenance is performed. They shall therefore be introduced for any kind of configurable behaviour to allow any custom tailored behaviour.

Policies can be inherented and applied to any kind of object. When applied to a schema all entities inherent that polcies. However, a different kind of policy with the same type can be applied to an entity overriding the one inherented by the parent. Policies are about background processing how metadata and constraints on different objects are enforced. They provide a lightweight version of UDFs (User-Defined-Functions) to build custom-tailored behaviours into the system. Other than the central configuration which is used to define core system behaviours. Such policies can be defined as:

Consistency Policies
Provide a notion of tuneable consistency, where users should be able to decide which levels to fulfill. In such a case an administrator could choose to define how many primary replicas an object should always contain. This would directly impact the constraints on the table restricting users to remove more placements than defined by that policy.

Freshness Policies
Can be utilized to define behaviour on freshness related actions. As [23] stated it is crucial to define how far a replica can diverge from the true and up-to-date value before a refresh transaction has to be critically executed. Additionally, we could use these types of policies to let object owners define how their data shall be identified and consequently updates are propagated. In such a way the system would stay customizable and would allow any methods discussed in **??** to be used dependent on the use case. Considering how refreshes are triggered one policy for example might has chosen to defer a propagation transaction entirely hence it won't try again and essentially waits for another chance when a new update-transaction is being executed and will trigger the service again. Another policy could suit other use cases better and assist by constantly querying the storage's performance metrics to decide if an update should be executed.

Due to the inherent heterogenous nature of the polystore systems itself, use cases may widely

vary. Hence, in gerneal there is no need to impose a general notion of freshness that is valid for all applications. Some might consider using CDC, while other prefer a partition approach or materialized views. However, with policies these can all be implemented.

Since applications that are being served by polystores are very different to each other, they might have different requirements. Therefore, different object types shall be supported.
A policy shall generally be created inside a database.

```
CREATE POLICY policy_name as <configuration>;
```

These policies can then be added to any object.

```
ALTER TABLE dummy ADD POLICY policy_name;
```

Users should always be able to list information on all applied policies on any object.

```
SHOW POLICIES ON (DATABASE | SCHEMA | TABLE ) object_name;
```

Furthermore they should be able to view the content of any policy:

```
DESCRIBE POLICIY policy_name ON (DATABASE | SCHEMA | TABLE ) object_name;
```

To complement the idea of policies we also need a central *Policy Manager* which ensures that the specified intentions are ensured. The manager shall be added as an additional central component and acts as a verification layer whenever meta information on objects will change.

7.2.2 Session-Wide Freshness

Another addiitona to freshness could be the extension to also allow the specification of freshness per session. This avoids specifiyng the freshness for individual statements. This is especially useful if the freshness requirements do not really change, allowing a quick possibility to adapt the requirements. Although, they could we extented for individual stateemntes, that indeed erquire a more strict form of freshness, it provides a good base line to operate on freshness. This is espeicially interesting for applications that usually establish one session, for the majority of its lifetime.

Essentially for every
configurabel freshnes
related paramteres a
with modification de
ation or time deviati
or if modification de
ation than total or p
entitiy

Add spacing in chap
to avoid entire block
data

Bibliography

[1] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, pages 37–42, 2012. doi: http://doi.org/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, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581026. doi: 10.1145/1376616.1376712. URL https://doi.org/10.1145/1376616.1376712.

[3] R. Agrawal, A. Ailamaki, P. Bernstein, E. Brewer, M. Carey, and S. Chaudhuri. The claremont report on database research. 52(6):56–65, 2009. ISSN 0001-0782. 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, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138598. 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, 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., 1986. ISBN 0201107155.

[12] E. Brewer. Towards robust distributed systems. *In Symposium on Principles of Distributed Computing (PODC)*, 2000.

[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, 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. ISBN 0897910737. 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. ISSN 0163-5808. doi: 10.1145/335191.335391.

[17] 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, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450307918. doi: 10.1145/2274576.2274589.

[18] 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.

[19] 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.

[20] O. Etzion, S. Jajodia, and S. Sripada, editors. *Temporal Databases: Research and Practice*, volume 1399. Springer, 1998.

[21] J. Faleiro and D. Abadi. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.*, 8(11):1190–1201, 2015. ISSN 2150-8097. doi: 10.14778/2809974.2809981.

[22] 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. ISSN 0163-5808. doi: 10.1145/2094114.2094126. URL https://doi.org/10.1145/2094114.2094126.

[23] A. Fekete. Replica freshness. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018. doi: 10.1007/978-1-4614-8265-9_1367.

[24] 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.

[25] M. Fowler. Polyglot persistence, 2011 (accessed April 14, 2022). URL https://martinfowler.com/bliki/PolyglotPersistence.html.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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)*, pages 26–37, 2013. doi: 10.1109/ICDE.2013.6544811.

[32] F. Naumann, U. Leser, and J. Freytag. Quality-driven integration of heterogeneous information systems. 1999.

[33] 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.

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

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

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

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

[38] D. Pritchett. Base An Acid Alternative. *ACM Queue*, 6(3):48–55, 2008. doi: 10.1145/1394127.1394128.

[39] 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, 2015. doi: 10.1007/978-3-319-15350-6_7.

[40] T. Redman. *Data Quality for the Information Age.* Artech House, 1996.

[41] 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.

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

[43] 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.

[44] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, and M. Ferreira. C-store: a column-oriented dbms. volume 2, pages 553–564, 2005.

[45] 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, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522731.

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

[47] 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.

[48] 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)*, 2020. doi: 10.1007/978-3-030-71055-2_2.

[49] 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, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-93662-4. doi: 10.1007/978-3-030-93663-1_6. URL https://doi.org/10.1007/978-3-030-93663-1_6.

[50] 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. ISBN 9780769541303. doi: 10.1109/CLOUD.2010.75.

[51] Y. Wei, S. Son, and J. Stankovic. Maintaining data freshness in distributed real-time databases. volume 16, pages 251 – 260, 2004. ISBN 0-7695-2176-2. doi: 10.1109/EMRTS.2004.1311028.

[52] 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.

[53] J. Xiang, G. Li, H. Xu, and X. Du. Data Freshness Guarantee and Scheduling of Update Transactions in RTMDBS. In *2008 4th International Conference on Wireless Communications, Networking and Mobile Computing*, pages 1–4, 2008. doi: 10.1109/WiCom.2008.1324.

[54] Y. Zhao and Y. Wang. Partition-based cloud data storage and processing model. In *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, volume 01, pages 218–223, 2012.

[55] 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.

[56] 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

Maybe also add MQ

This chapter provides an in-depth extension to the existing PolySQL Syntax for freshness related queries. The original PolySQL Syntax will not be illustrated in this chapter.
All valid extensions for Freshness must consequently begin with the keywords *WITH FRESHNESS*. They are attached as an optional leaf expression for every *SELECT* statement.

A.1 PolySQL

```
1   SELECT * FROM tableName
2   [ WITH FRESHNESS
3       [
4           (
5               TIMESTAMP
6               |
7               <DELAY>
8               |
9               <INDEX>
10          )
11      ]];
```

A.1.1 Absolute Timestamp

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

A.1.2 Relative Timestamp - Absolute Delay

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

SELECT * FROM dummy WITH FRESHNESS 3 HOUR ABSOLUTE;

SELECT * FROM dummy WITH FRESHNESS 3 MINUTEs ABSOLUTE;
```

A.1.3 Relative Delay

```
SELECT * FROM dummy WITH FRESHNESS 3 SECOND DELAY;
```

```
SELECT * FROM dummy WITH FRESHNESS 3 HOUR DELAY;

SELECT * FROM dummy WITH FRESHNESS 3 MINUTEs DELAY;
```

A.1.4 Freshness Index

```
SELECT * FROM dummy WITH FRESHNESS 0.6;

SELECT * FROM dummy WITH FRESHNESS 60%;
```

A.1.5 Refresh Operations

```
ALTER TABLE dummy REFRESH ALL PLACEMENTS;

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

B

Query Templates for the Benchmark

For better reproducibility this chapter contains all utilized query templates for evaluating freshness-aware data management within Polypheny-DB. As well as the procedure on how to carry out those tests.

C
Evaluation Results

This appendix lists all acquired plots and evaluation results that have been conducted and summarized in Chapter 6.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

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

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 .