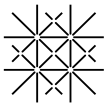

University of Basel

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

For the past decade, cloud computing has become a crucial and central part in the industry [3]. This technological advancement in infrastructure provisioning allows companies to obtain software as well as hardware resources to compose their entire data center virtually in the cloud. Without having to invest in expensive hardware and real estate they can now avoid maintaining their computing resources solely on premise. These providers usually manage different distributed data centers across the globe and provide private or shared access on resources according to their Service Level Agreement (SLA). Such guarantees in quality-of-service (QoS) usually include elastic up- and down-scaling of resources and a high degree of availability, which is achieved by replicating data throughout different regions [13] [44].

Ever since the rise of the *Big Data* era, these providers are faced with continuous and rapidly growing heterogeneous datasets. These are accompanied by widely varying requirements and characteristics for data driven applications. This increased the need to leverage custom-tailored database systems to gain meaningful insights on their data as quickly as possible. To efficiently process and extract relevant information out of these data silos new systems have emerged. These aim to provide a solution for the new demands on varying workloads and ever-growing data sources. Such novel Polystore systems combine several distributed physical data engines to enable various new possibilities and provide the best response time for every use case by exploiting the key benefits of each engine [42] [47]. Although these data management systems are inherently built to process heterogeneous data with high throughput, the amount of produced data still grows continuously. Therefore, the importance to efficiently access the right data is crucial for organizations to stay economical and competitive. Cloud providers which offer such systems consequently need reliable functionality to manage these large volumes of data. Otherwise, they would waste useful computations and time when users try to retrieve relevant data items [30]. To comply with their QoS requirements and sufficiently provide acceptable access times for their services, they have come up with data freshness strategies as an essential part of data management in a distributed setup. The freshness in such cases reflects how current and up-to-date a specific data item is. Since not all applications pursue similar goals, they often require different levels of freshness. These levels can be used to identify a well-suited location to retrieve the

desired data object. This consequently mitigates the need to always update every existing data replica to the most recent state. Changes can now rather be deferred, until they are processable by the underlying system again. Such delayed updates now allow for a much higher throughput and increased performance in scenarios were also slightly outdated data is acceptable.

1.1 Goal?

The system therefore needs to be able to allow transactional as well as analytical workload in parallel.

1.2 Contribution

The contribution of this thesis is fivefold. First, we identify and define necessary requirements to establish freshness-awareness in general. Second, we outline and propose various possibilities to enable freshness-awareness. Forth, we introduce a query extension to allow the specification of tolerated and desirable freshness levels on various metrics. Fifth,
To identify necessary requirements to implemented and design possible freshness variations in a polystore system. Several aspects of previous published research work is compared and considered when adapting it to polystore systems.

1.3 Outline

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

Missing: TCO becau
even if highly replica
systems we might on
use the secondary no
in case of failure situ
tions.

Missing: with
Polyphenys lazy repl
cation we could limit
the primary update
to fast OLTP driven
stores and lazily upd
ing other stores

Missing: Allow para
workload (transactio
analytical) by reduc-
ing locks and allowin
reads to be applied t
outdated nodes

2
Motivational Scenario

Explain scenario and why especially necessary on polystore systems transactional UPTO-DATE workload on postgres and Analytical in-memory order column-oriented DB.
(Imagine a distributed system)
COnsider locking in a distributed setup. Why this is limited by the slowest performing node.
(maybe cite) Amdahls law .

This might not be su able in Motivational Scenario

We can now harvest the benefits of a polystore system with their distinct replication engines
Imagine an *Enterprise Resource Planning* (ERP) system, containing several different customers. This comprehensive system is able to provide purchasing, accounting, inventory stocking and warehouse management within one system. Since these systems are inherently used to reproduce an entire supply chain, they can be easily be used to report on different sales metrics or run analytical queries to aggregate certain multidimensional results. This is certainly desirable to analyze the current state of the company as well as adequately adapt to new trends or opportunities. However, even in we imagine that we have a distributed system
If our system however only supports strong consistency by eagerly replicating all incoming updates to every available node, we are limited by the slowest performing node in this setup. During these write operations no could
Explain that this system in this scenario does not allow concurrent writes and reads. Since reads do not alter that data, we can provide multiple

Missing: Cite this

However, since ERP are strong transactional systems receiving write-heavy workload the write speed is essentially bound by the lowest performing
This does not only limit locking situations but also increases the databases' response time. Furthermore, this enables this system even in a distributed setup to work with several nodes. Even if not all nodes are directly notifying the system that they have succesfully finished the operation we could continue with the operation. Depending on the tolerated level of consistency we give in on consistency to improve availability. This can of course also be tuned to wait until a majority or defined number of nodes have committed the transaction.
All activity is primarly executed through a central ERP system.
Since this company has locations around the world the ERP system has an evenly distributed load throughout the day without any peak performance windows. The availability as well

Missing: Imagine a global manufacturing company that is responsible for the ent supply chain. enable direct purchase throu an ERP system whic has several thousand customers, productio ready assembly lines

as the processing throughput of this system is crucial for the company to stay competitive. While the processing department concerned with sales and production, the business intelligence (BI) team on the other hand needs to analyze to provide sales forecasts recognize upcoming trends. Since the company and data is still globally distributed and each entity additionally stores some information locally, that are unique to this branch or devision. So any analysis done by the BI team needs to retrieve data from different locations maybe even companies. Although, these They cannot interfere with the daily-operation of the company. Although this is provided within one central system the use cases and requirements are fundamentally different. Since the company is globally distributed among several locations the data storage engines are distributed across the globe. Since this is a central system the availability as well as the consistency is important, to protect itself against failures while still providing an acceptable throughput.
For the sake of simplicity any legal constraints on archiving financial data are omitted.

3

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 3.1), possible metrics to define cost functions which are used in the evaluation process as different freshness-levels are expressed and compared (→ Section 3.2), how to involve different versions of data items (→ Section 3.6). Furthermore, which possibilities exist on how to replicate data (→ Section **??**) and propagate updates to outdated nodes to refresh them. Finally, this is concluded how users can actually specify their tolerated levels of freshness (→ Section 3.7).

Missing: Cite from S entific Writing repor [27]

Missing: Cite from Marco Dis [45]

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 [54]. 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 [37]. *BASE* in this context stands for **B**asically **A**vailable **S**oft State and **E**ventual Consistency [41]. 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. [49] 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.

3.1 Data Freshness

The idea of freshness is a widely used concept among distributed data integration systems and intuitively introduces the idea of how old or stale data is. However, several notions and interpretations exist, on how to characterize and measure data freshness. In 1999 Naumann et al. [31] already stated that although the idea of data quality and freshness has no commonly agreed definition and varies among use cases, it is strongly related to the concept of accuracy. Such accuracy of a data object can roughly be summarized as the percentage of objects without data change. This thought was also shared by the author of [39] 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 therefore be considered as the precision and accuracy of such data elements in respect to its most up-to-date version. 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. Missing: cite
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 [35] also described freshness as a matter of how old data is and linked this to an user's expectation who is interested when the data was produced or if there may be other sources that have more recent or different kinds of freshness. Consequently, using the last time objects were updated, to identify any accuracy measures. Peralta also distinguished between two quality factors. The *Currency factor* which expresses how stale data is after it has been extracted and delivered to the user. This concept is often considered in Data Warehousing systems, when already extracted materialized data is processed and delivered to the user while the source data might have already been updated after it has initially been extracted and materialized resulting in stale data. The second factor corresponds to the *timeliness* of data. This essentially states how old data is and captures the gap between the last update and delivery.

3.2 Freshness Analysis and metrics

A metric is a specific figure which can be used to evaluate given quality factors. With the various definitions and approaches to data freshness itself, the metrics to identify and measure the level of freshness also vary. Several metrics are summarized by [16, 33, 35] 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 updates that still need 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.

While [6] and [53] define the notion of freshness by relating it to the *age-of-information* (AoI) respectively the timeliness represented by the timestamp of the transactions which has updated the value has as well as the *age-of-synchronization* (AoS) which corresponds to the time when the value was updated.

3.3 Freshness Constraints

Along the various freshness metrics as described in 3.2, Tamer et al. [54] list three types of freshness constraints that users could specify to suggest an accepted level of freshness.

- **Time-bound constraints**: Are used as timestamps that tolerate values which were updated by younger transactions.
- **Value-bound constraints**: Consider percentage deviations from the current value. This is more commonly used with numbers than with text and is considered to be mutually consistent if the values do not diverge more than a certain amount.
- **Drift constraints on multiple data items**: Relevant for transactions which read multiple data items. It is acceptable for users if the update timestamps of all 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 [49]. 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

$$f(d) = \frac{t(d)}{t(d_0)}, with f(d) \in [0, 1] \tag{3.1}$$

resulting in a *freshness index*. Röhm et al. [40] 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 [51] and [22] 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.

3.4 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. 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 secondaries exist which hold slightly outdated and 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 [38, 40, 51] have pointed out in their work.

Missing: cite existing

Missing: cite

Depending on the architectural setup, different refreshment strategies have been proposed on how updates are propagated towards outdated nodes.

Wei et al. [50] propose a replication in form of an update policy adaptation algorithm that dynamically selects update policies for derived data items, This is fulfilled according to system conditions to have several layers of validation transaction to be fulfilled to actually be executed. 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 since the transaction itself could be handled multiple times before it might actually get executed.

Rewrite

Psaroudakis et al. [38] mention the existing design gaps between OLTP- and OLAP-oriented DBMS and describe the importance of allowing the execution of OLAP queries even during the execution of transactional workload. However, they merely focus the 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 described the idea of *SAP HANA* to offer a main area which contains the base data and a delta area which supports transactional operations and includes recently changed data. Read operations need to query main and delta 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 [50] which aimed to mitigate the complexity and replication overhead by combining base data elements and derived elements. For queries, a base set of information is enriched with delta fragments to derive the relevant output. This reduces

the needed update cycles on all replicas since base information is always available and queued changes are applied during runtime to recompute the actual response. However, this approach is costly 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. [34] introduce concurrent replica refreshments and discuss the idea of *Preventive Replication* by using an asynchronous primary-copy replication approach and 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 [49] 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 updates acting as primary servers, level-2 sites are read-only nodes containing as up-to-date data as possible and level-3 read-only nodes where data is updated rather infrequently. These levels are layered and composed as a tree receiving asynchronous updates from the lower level. The higher the level, the more outdated the data.
In contrast, [51] approaches its replication algorithm with a relationship between data items by utilizing 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 multilayer replication setup.

3.5 Refresh Strategies

Based on the discussed lazy propagation mechanisms in 3.4, three refresh and update strategies could be identified when updates should be propagated to replicas.
Although the authors of [38] rather focus on table-level objects and not on replication scenarios the up-date mechanism still follows the same requirements to jointly support a mixed workload of OLTP- and OLAP-oriented transactions. It considers a periodic approach when the main and delta areas shall be merged which essentially refers to updating an outdated base item, respectively the main area.
Others [40] avoid the scheduling of such update propagations completely by simply decoupling the primary update transactions from the read-only nodes and immediately executing the propagation-transactions as well. Although this helps the throughput of the initial update transactions the secondaries do not really stay outdated for any time at all. Since, such approaches as well as periodic scheduling completely neglect to verify the load on the underlying system it could cause unnecessary resource consumption on the replicas to be updated. Therefore, this should always be accompanied by identifying the idle time of replicas as well as ensuring that the current load on the replicas is not too high and is therefore able to endure such an update [49].
In contrast to [35] which talks about determining 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. Any outdated node will independently pull the

number of updates which are necessary to fulfil future requests of such requirements.
Wei et al. [50] follow a slightly different approach by analyzing the *Access Update Ratio* (AUR) for any data object **d** which is given as:

$$AUR(d) = \frac{AccessFrequency(d)}{UpdateFrequency(d)} \tag{3.2}$$

. Consequently any item above an AUR of 1 is considered to be accessed at least as frequently as it is updated and 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 takes into account that this is accessed rather infrequently and thus needs no immediate refresh.

3.6 Consistency

There are several constraints to consider which ensure the consistency in a distributed setup. The authors of [50, 51] considered data freshness together with the scheduling of update transactions and talked about its freshness from the point of view of real time applications. They considered the concept of temporal validity in context of value-based freshness, such that specific values are only valid for a certain time interval before they become outdated. Hence, it is necessary to classify objects into temporal data that can continuously change to reflect a real world state and non-temporal objects than 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 reconstruction of any past value [19].
Voicu et al. [49] propose a decoupling of transactions to correctly separate the individual requirements for update propagations. These transactions are differentiated in regular update transactions that target primary nodes, propagation transactions that refresh read-only nodes, 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 [38] introduced a common data structure for both workloads for delta and main store and the usage of multi-version concurrency control (MVCC) to provide access for both OLTP and OLAP. This implicitly enables the system to keep several versions of a data item (see Section 4.5). These versions can be directly used as outdated or freshness guarantees. However, the implementation of MVCC is complex and needs more system resources. Although this would indeed improve update times and would support referential integrity.
Finally, the authors of [5] propose freshness locks which are applied to the stores which have been selected to fulfil the read requests. These locks aim to provide fast response times for OLAP workload and ensure that data is not refreshed when being read within one transaction.

3.7 Freshness-aware Read Access

Since the essential idea of data freshness revolves around fast read access while transactional workload is currently being executed, 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 [40] a router needs to utilize system state information on replicas to identify the correct way to route a query.

In [49] clients can directly contact any read-only replica. For every read-only transaction they can as well specify their tolerated freshness level, by specifying 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 able to identify which node is able to fulfil the condition. If none of the nodes are able to fulfil the request in this subtree a refresh transaction is executed updating all nodes before processing the read operation.

The authors [28] 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 and by comparing latencies for any request 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 and refers to the average time how much time passes before an update is propagated to the secondaries.

4

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.

4.1 Polystores

The decision which data structure to use and built upon is a crucial step for the overall performance of a systems design, as suggested by H.Plattner and B.Leukert [36].
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 nowadays there exist a variety of Database Management Systems (DBMS) 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* [47]. Polyglot persistence is a term which refers to a practice originated from the concept of *polyglot programming* or microservice architectures, to utilize different programming languages for different task requirements following a best-fit approach [24].
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.

4.2 Data Partitioning

Beside the utilization of several data storage engines, the data model and structure will have an enormous 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.
In general data partitioning can be distinguished between two variations. Both of these forms split the data into multiple parts. It is therefore often called fragmentation . Missing: Cite

Vertical Partitioning is usually applied during the design of a data model inside a database. This involves the creation of tables with fewer columns and therefore using additional tables to store the remainder of columns [32]. This approach is often used in the context of normalization of a data model [17]. In order to combine and reconstruct these vertical partitions again there needs to be some sort of redundantly stored data like the primary key. Which uniquely identifies individual items.

Horizontal Partitioning refers to the partitioning of objects like tables into a disjoint set of rows that can be stored and accessed separately [15]. To support this explicit form of partitioning there exist several partition algorithms. The most common ones are List, Range and Hash partitioning. These algorithms can be applied to a table based on an arbitrary column which results in a fragmentation of the table based on 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 DBMS 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, which is still one of the main pursuits of current data management solutions [14].

4.3 CAP Theorem

An essential part of distributed systems is the handling of failures or outages of participating components. The *CAP theorem* [25] 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.

4.4 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 [52]. 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 [29]. 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 4.3.
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 [26].
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 [22]. 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 [18] However, during the convergence period, until all changes have been replicated, the system is exposed to an in-

consistent 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*. 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. [29]

4.5 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 4.3, 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).

4.5.1 SS2PL

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* 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* 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.

4.5.2 MVCC

While *SS2PL* suffers from lock contentions between long-running analytical reads and update transactions, it cannot support parallel writes and reads. 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 *snapshot isolation* which was introduced by Berenson et al. [7]. This enables each transaction to see the state of the data at the time when the transaction has started.

Since snapshot isolation provides weaker consistency guarantees as *one-copy serializability* (1SR) it can increase the concurrency by relaxing the isolation requirements . Despite the existence of multiple versions on several replicas that provide more flexibility for the scheduler, it is more difficult to find a corresponding serializable schedule in accordance with 1SR according to [18].

Although, there exists some work that aims to provide serializable versions of multi-version concurrency control [20, 23].

Moreover, MVCC protocols are challenged with the decision how many version to retain and when to old ones become obsolete and can be removed. Generally this leads to a larger data footprint since objects are inherently stored redundantly.

Missing: Talk about append only table w large data footprint

Missing: Remove Co rectness Guarantees

Missing: cite

Missing: Add Write Skew anomaly?

5 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 3 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.

5.1 Functional Requirements

Based on the provided discussion in Chapter 3 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 (→ 5.2), 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 (→ 5.3) and further provide users with a possibility, to formulate an acceptable level of freshness (→ 5.7). Based on these fundamentals we need to consider how update transactions can be decoupled and defer the refresh of specific replicas (→ 5.4), all while ensuring the consistency of the entire system (→ 5.5) to finally improve the query routing to identify freshness levels to increase the performance of read-only operations.

5.2 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 3.5 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 5.4.2.

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, 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.

5.3 Freshness Metrics

As discussed in 3.3, 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 5.1. 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.

$$F(d, \delta) := \begin{cases} \text{true} & \text{if } f(d) \geq \delta \\ \text{false} & \text{otherwise} \end{cases} \tag{5.1}$$

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 5.2, 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.

$$f(d) := t(d) \tag{5.2}$$

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 5.2 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 5.3 will essentially compare the current commit timestamp $t(d)$ against its primary replica $t(d_{primary})$.

$$f(d) := t(d_{primary}) - t(d) \tag{5.3}$$

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 [40]. 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 5.4.

$$f(d) := \frac{m(d)}{m(d_{primary})}, with f(d) \in [0, 1] \tag{5.4}$$

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 [27, 49] 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.

$$f(d) := \frac{t(d)}{t(d_{primary})}, with f(d) \in [0,1] \tag{5.5}$$

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 5.2 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 3.7). 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.

5.4 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 5.2 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.

5.4.1 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 modifi-

cation 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 5.5) 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.

5.4.2 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 5.4.1 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 possibly existing functions or constraints might have already been evaluated at runtime. Thus, we can save further computations and store the end result that is pushed-down directly to the designated stores. As soon as this transaction has been successfully committed, all entries in this capture-queue are further enriched with the respective commit timestamp of their parent transaction. Afterward the corresponding entries in the capture-queue are added to an actual central replication queue containing the pending updates. For each designated replica that should receive this update an individual entry is created inside this queue. Each entry is accompanied by its parent transactions ID, the commit timestamp as well as the data to be replicated. Since we do not need to store the

data n-times for n replicas determined to receive the data, we can simply link each replication item in the queue to its corresponding replication data, which is stored separately. Since all entries in this final replication-queue are ordered with respect to the execution order of the original transaction, we have ensured that the operations are executed in order to converge to the same state as its primary copy.
Finally, if the transaction aborts, all active entries in the initial capture-queue can be removed due to their association with the parent transaction.

Primary Data Snapshot Although CDC will correctly recreate any secondary replica, it will lose its efficiency when there are more 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 5.4.1, 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 5.2 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.

5.5 Consistency Constraints

As suggested in 3.4 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 5.4.2 we require a *Refresh-Lock* as a newly introduced lock. 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.

5.6 Transaction Handling-Constraints

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 5.2 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)*.

5.7 Freshness-aware Read Access

As mentioned in 5.6 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 5.3 and the different available versions per data object (see 5.2) 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 3.7) 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 [49].
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 iteam 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.

6 Implementation

This chapter describes an implementation in correspondence to the concepts proposed and ellaborated in chapter 5. 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 structured as 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.
Change Data Capture (→ *section*??) Replication Algorithm(→ *section*??) Freshness specifications(→ *section*??) Freshness evaluation(→ *section*??) Freshness constraints(→ *section*??)

Talk about each implementation step fo the building blocks i general, but if there deviations e.g. in lan guages briefly differe tiate them with bull points and add them the appendix

6.1 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. This extends the foundations laid out in Chapter 4 and sets them in context of the existing system model.
PolyDBMS [48]
Polypheny-DB is an Open-Source project[2] developed by the *Database and Information Systems* (DBIS) group of the University of Basel.

Add PolyDBMS cite Love Marriage or Ma riage of convenience

Polypheny-DB is a self-adaptive polystore that provides cost- and workload aware access to

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

heterogeneous data[46].
Compared to other systems like *C-Store*[43] or *SAP HANA* [21], Polypheny-DB does not provide its own set of different storage engines to support different workload demands.
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.

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

6.1.1 Placements

Placements are considered to be Polyphenys virtual representation of physical entities. 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.

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 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

It contains information on available columns (→ Column Placements), partitions (→ Partition Placements) as well as properties unique to this store.

A table can therefore contain several Data Placements with different capabilities and properties.

Missing: Image

are used along the idea of Column Placement. A Data Placement is a representation of table with all placed columns on a specific physical store.

When a table is created on Polypheny-DB it is an ordinary structure placed onto one store. Such a table 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 in form of a *Data*

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

Placement. This can either be done by evenly distributing the columns onto these stores or by simply replicating the subset to the second store.

Column Placements are needed to fulfill the intended flexibility of Polypheny-DB. Column Placements are instances of a column placed on a specific store. These placements are the result of the extended vertical partitoning of a table.

Missing: Image

Column Placements are instances of a column placed on a specific store. These placements are the result of the extended vertical partitioning of a table. They are considered unique per column on a cluster.

Missing: Maybe summarize this under Data Placement

As already discussed in 4.2, 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. This extension provides the functionality to place columns rather freely on a store without replicating the complete table. Although these columns are logically bound to a table there is no need to replicate the complete structure to a desired store. In some cases this does not only result in an optimized access of the data part but also saves data overhead on the specific store.
This functionality enables Polypheny-DB to adapt the data structure to continuously varying use cases.

Partition Placements Due to the partition function NONE every table 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 or list().

A Partition Placement is

6.1.2 Query Routing

Missing: Image

Routing or cache plan optimizat execute
Since every query has to go through the abstraction layer to guarantee correctness and consistency, Polypheny-DB can consult the systems *Catalog* to retrieve the location of all relevant data. This is done by gathering all *Column placements* needed by the query.
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.
Since partitions are mere logical identifiers there main usage is to locate data fragments or the location where a query should route a statement to. One physical table can therefore hold several partitions. During routing if a partition has been identified it is checked for every store involved whether it contains and associated partition placement. Although, this routing method is quite fast there are several problems concerning the separability of data. This imposes for one the difficulty to retrieve data belonging to exactly one partition out of a

table which contains several partitions and secondly difficult to migrate data from individual partitions to another store.
Since it is rather complex to extract all relevant partitions needed for a specific query especially when combining vertical and horizontal partitioning the concept of *Worst case Routing* was introduced.
This routing mechanism aims to improve performance, when the process of identifying the correct partitions for a query would be too complicated and therefore also reduce the overall performance. This is the reason why currently, a *Full Placement* has to be enforced for all three Partition Managers to support the functionality of worst case routing. A Full Placement in that sense refers to a placement on a store which contains all partitions of a specific table and can therefore be used as a fallback scenario.
However, due to the adjustments to partitioning including the new Partition Placements which are represented by their own individual physical tables, the constraint imposed by logical partitions and therefore the necessity of a full placement per table can be removed.
Queries are now able to flexibly combine vertical and horizontal partitioning to truly leverage the power of Polypheny-DB. The routing for SELECT-queries is now simplified since it aims to find for each partition all requested columns by joining Column Placements per partition first and then applying a UNION over all accessed partitions to build the required result set.

6.1.3 Concurrency Control

Given Polyphenys current architecture all incoming queries has to be delivered through the poly-layer, 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.

Missing: Maybe rename?

6.2 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 a mixed worklaod to exist in parallel.

6.2.1 Placement Versioning

As we have established in section 5.2, 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 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. But, this would also imply the utilization of MVCC. However, as stated in 4.5 multi-version databases automatically tend to have larger data footprints, due to persisting redundant and even obsolete data. Howeever, polystore systems already suffer from a larger data volume, given the redundant data storage across severall stores. Finally, 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 5, multiple versions are automatically created when using a lazy update propagation. This will directly loosen the constraints, impossed 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 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 4.2), Polypheny-DBs partition placements are suitable candidates to base our freshness awareness on.

Missing: Besserer Übergang

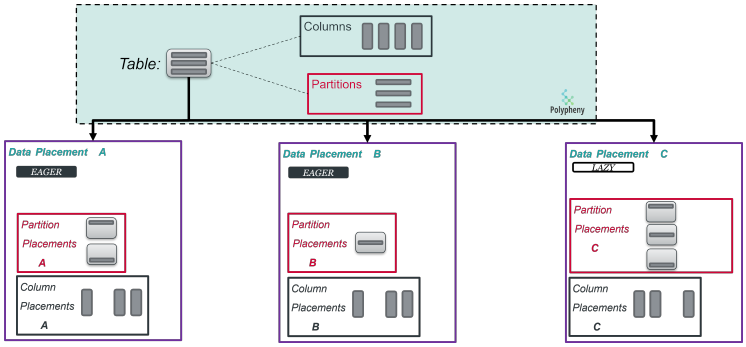

Figure 6.1: Entity alongside all its physical placements

6.2.2 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.

Missing: integrate SYNC and ASYNC

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 WITH REPLICATION
      ( LAZY | EAGER ) ;
```

This replication strategy is added as part of the newly added 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 6.2.8).

6.2.3 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. 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 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. Since the internal partitioning should be rathe ruser 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.
They have a dependency that refreshable can be set to INFINITELY OUTDAED and vice versa, however UPTODATE can only be influenced by the replicaiton strategy. Trying to change this manually will result in an error since it is controlled by the system.

Missing: flowchart?

Placements containing *REFRESHABLE*

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

OUTDATED referring to INFINITELY OUTDATED, by explicitly labeling it as outdated it will suspend all update propagation towards those stores.
Furthermore each Partition Placement is enriched with the most recent update information to support various freshness metrics. This update information contains insights such as the id of the last comitted update transaction, the corresponding commit and update timestamp as well as the number of modifications that
Therefore, 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.

6.2.4 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.

After creation this object will be added along its statement id to a preliminary *capture-queue* within the *Change Data Collector*. As visualized in **??** this capture-queue is represented as as 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*.

Missing: Add illustration of hashtable in data colelctor identifiyng transaction to stateemnt.

To be able to apply operations directly to the outdated replicas, they need to be converted into basic operations that can be applied to a prepared statement. Therefore they are captured before they are executed but after they have been evaluated.
Since not all stores provide the same functional capabilities, we can leverage the polystore-layer to pre-compute certain calls before applying them to the underlying stores. Typical functions that are not uniformly provided are e.g.: $CURRENT\_TIME$ or $TIME\_NOW$. This allows storing the actual values that are executed on the store, hence saving execution time during update propagation. During runtime of any given statement the actual evaluated data values are then injected into the object stored within the capture-queue.

The benefit of this structure is that as soon as the transaction commits, the *Change Data Collector* is notified, streaming all objects in correct execution order into the *Replication Engine*, where they will be transformed into individual *Replication Objects* and finally queued to be propagated onto outdated placements. Since the registration is done during the commit, we are sure that any pending changes will be available for distribution once the transaction has been commited.

6.2.5 Lazy Replication Engine

Is build on a basic *Replication Engine*, that 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 5.4.2 and is influenced by the change data capture service of section 6.2.4.

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 replication 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 been executed on the primary placements. Therefore it keeps information on the orginal transaction, its commit time as well as the operation type and the data to be delivered. This is further enriched with a list of all target partition placements that shall receive this modification. The list of targets is generated at the time the initial update transaction has been committed and changes have been queued. In order to avoid storing data redundantly, this data object is centrally stored and only referenced at the depending replication objects disregarding the number of placements that shall receive a given DML-operation.

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. It is represented as a FIFO queue receiving new replication IDs that are resitred through the *Change Data Collecor* or rescheduled from workers.

As depicted in figure 6.2 this

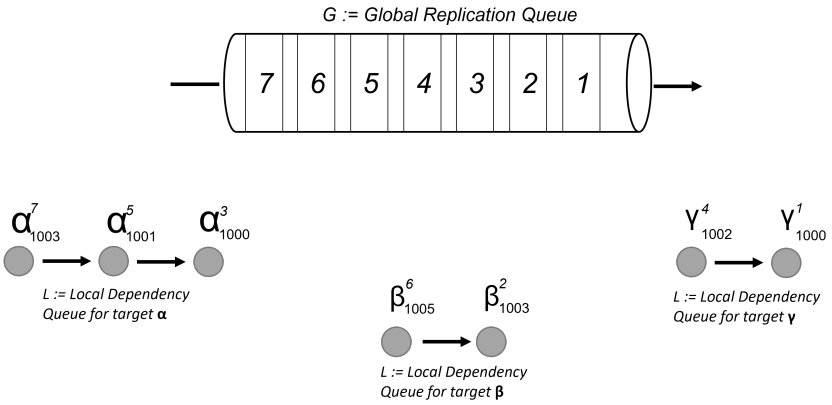

Figure 6.2: Global and local queue

Each replication event within the queue is therefore defined as x_z^y, where x is the designated target partition placement, y a global replication id uniquely identifiying a specific replication and z 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

Local Replication 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 in their execution order, we are certain that the imposed dependency reflects the original execution order. Therefore this queue is used as a utility to enforce constraints and ensure that operations are applied in correct order.

Replication Worker These workers are an essential part of the replication engine since they continuously process events from the global queue.

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 Replication Queue* always will ensure the correct execution order a concurrent processing of the replicaitons is possible.

Data Replicator Is implemented as the

Update Metadata Since data is essentially stored on physical entities, represented by the internal partition placements, we extended these objects to contain information on general update statistics relevant for this particular placement. This 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 placment has already applied. Allowing us to use those metadata to compare different versions against each other.

Missing: Maybe illustration

- Queueing process as flowchart - replication process as flowchart

Missing: Change D Capture -Is separate into 2 phases

6.2.6 Automatic Lazy Replication Algorithm

Missing: In correspo dence to the section update propagtion i concept we focussed implementing the C approach (→ 6.2.6) well as the on-dema approach using Pri- amry Snapshot Copy (→ 6.2.7))

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 , 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 enabels 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.

Uses the provided change data capture approach

ref image of flowchar

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).

add link to image

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 fottprint 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 placments, 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 transaction 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 priamry execution.

6.2.7 Manual Refresh Operations

Despite that the autoamtic refresh operation will take care of most the occuring changes, users might need to be able to specifically priorities certain data placements or entire tables to be updated faster than others. Additionally 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 5.4.2 we need to be able to provide manual refreshes on the basis of primary snapshot copies. This inherently uses the capabilities of Poylphenys existing *Data Migrator*, which essentially queries a defined source entity on a given store. and subsequently will apply this data onto a targeted placement. This can therefore be used to for snapshot copy.

```
ALTER TABLE dummy REFRESH PLACEMENT ON STORE outdated_store;

ALTER TABLE dummy REFRESH ALL PLACEMENTS;
```

Although such refresh-transactions can be executed on any placement, it will have no effect on primaries and simply omit their execution.

6.2.8 Placement Constraints

Furthermore 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 distributed 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 ... in violating 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 tehy 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

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

Missing: Explain potential optimization steps that we can an alyze the queue and can aggregate certain steps or avoid certain operations if we exec batch wise and one U DATE operation e.g. updates the same pr mary key

Missing: Not specifically in manual, alte ing or inhernetly mo fying a data placeme with DDLs leaves th as INFINITELY OU DATED

Missing: Or is this tomatically detected since we centrally st the replication objec and can apply it to a bitrary placements d regarding their local columns or partition

Missing: LazyWorkerThreads - were eac

strategy internally. Since this is merely done by setting a flag, the impact of the blocking behaviour can be neglected.

Finally, as stated in 6.2.3 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 replication again we will lose data in between on this replica. 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 6.2.7. An immediate chnage of states will then be accepted.

6.2.9 Placement Refresh

6.3 Freshness Awareness

This section focusses on all aspects to establish the core aspects of freshness within Polypheny-DB.
Moreover, to improve the user experience the query tree paths should be extended to specify which level of freshness was requested and with which level it was ultimately fulfilled. Furthermore, for successfully executed queries that considered some kind of freshness it shall be added in the SQL response that is has been executed by means of a specific freshness level. In such a way a user can not only steer its desire but is also informed whether the request could be fulfilled.

6.3.1 Fresnness Evaluation Types

maybe put this as description item into another sub section

6.3.2 Freshness Query Specification

This can mainly be achieved by extending the query functionality of Polypheny-DB. The selection is able to facilitate all possibilities as provided in section
Although Polypheny-DB provides multiple query interfaces and languages, the following specifications are solely demonstrated with SQL. As proposed within section 5.3 we have introduced essentially three freshness types that can be used to specify a tolerated level of freshness.
Generally all freshness metrics as described in the corresponding sections and along the fucntional requiremnt *(ii)* have also been introduced within Polypheny-DB.
The freshness specification can be appended on any query operation. For SQL it is extended as an optional leaf expression at the end of every query.
For the overall freshness guidance an extension of PolySQL is necessary. Along the description defined users can choose to select any of the specifications to guide the system.

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

Also use simply *WITH FRESHNESS* to specify that you dont care with what freshness it returns

Missing: add this information also at the beginning on lazy replication, when explaining change data capture

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 also always executed by comparing partition placements. This is mainly done using metadata within the update information properties of each partition placement. These already contian information on the current commit and update tiemstamp, the original parent transaction which has updated this placement as well as the numbe rof modifications this particular placement has received.

Missing: relocate to freshness filter inside freshness manager how the filter operations are going to be executed

For every freshness filter we need to first analyze which partitions are needed for this query. This is done regardless of using freshness or not. Then the *Freshness Manager* retrieves all available partition palcements for each partition that are lazily replicated. Afterwards, these lists of partition placements is filtered based on the provided specification.

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 5.3, 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 priamry placement compared to an outdated placement. To differentiatet those options they are individually suffixed $\in \{ABSOLUTE, DELAY\}$

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. However, this index can also be configured using the freshness index to evaluate the time deviation between two replicas **Time Deviation**.

Essentially the index dictates how accurate a given placement is with respect to its up-to-date replica.

After the specification it is enriched

6.3.3 Freshness Detection and Extraction

For every query the query tree is parsed and evaluated in terms of syntactical correctness. Additionally, teh query is analyzed semantically where also the freshness will be extracted. This helps to autoamtically enrich the statement with the correct freshness inforamtion and informs the transaction that freshness is being used which influences locking capabilities and changes the routing process. The *Freshness Extractor* also checks if for the specified freshness evaluation type

6.3.4 Freshness Manager

This can be used if a user does not require a minimum level of freshness. This can be easily supported

Applies filters and co
bines different possib
ties

6.3.5 Freshness Filtering

This is done based on thedifferent freshness evaluation types.

Which possibilities
do we have to select
a query with freshne

6.3.6 Freshness Selection

Lifecycle how does the freshness selection process work in geenral. How is filtering combined.

How does the freshn
respresentation look
like for SQL.

6.3.7 Referential Integrity - Freshness Isolation

The new isolation level provided by Polypheny-DB in association with freshness can therefore we distinguished between three different levels. Since we already violated the ACID constraints by returning outdated data, the system initially treats every freshness-related wuery to not acquire any locks on outdated palcements, hence allowing dirty reads. This results in refresh operations being able to override or add new results to the placement while it is currently being read. While this is not harmful for consecutive write operations that are lazily replicated onto this placement, it could return entirely inconsistent data when an *INFINITELY OUTDATED* placement is being refreshed by the data Migrator. Since this empties the table and starts from the beginning. Therefore another possibility to avoid those dirty reads is to acquire locks even for outdated or freshness queries. Although the results stay stable and consistent in regards to the results provided, it will block refresh operations on this placement entirely. Since polypheny-DB uses a version of 2PL shared locks can be easily extended with more transactions adding itself to the list of waiting transactions, which could lead to starvation of the refresh transaction. 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 the

Missing: When a re
fresh transaction is e
cuted no more locks
secondary nodes can
applied, if there is st
a shared lock on tha
outdated node they
first be commited be
fore the refresh takes
place (avoid deadloc
if we do not hinder t
system to build more
shared locks refresh
erations might starve

WITH FRESHNESS

partition placments
filtered and compare
although as stated in
constraints, the prim
and lazy palcement
not have to be equal
terms of there repre-
sentation (can contai
different columns) ,
therefore just require

possibility for any new upcoming freshness transactions to add itself to the existing shared lock. Instead they are treatetd and added to the dependency graph of the refresh operation (requiring an exclusive lock) as if they woukd already be an exclusive lock in place. This would avoid straving the refresh operation while still being able to sevre freshness queries on different placements. This can also be enhanced by centrally configuring a threshold after how many retries the refresh operation can finally force itself to be executed. Finally we allow to specify a referential integrity enforcement. As described in section 5.7 we would allow this to try enforcing the referential intergity, within this transaction. Which also would need to lock the placements. This approach aims to enforce the refernetial intergity among all entitite sused within the transaction. However in an outdated scenario this can only be ensured if all entities, even have outdated versions of itself and they have all been updated by the same transaction and consequently. Otherwise we cannot easily guarantee that they enforce those constraints. 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 blocks updates on the priamry placements, it fulfils the constriants as it would have when executing the same query without specifying a tolerated level of freshness.

Missing: implement

7 Evaluation

This chapter is separated into

7.1 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

7.2 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.

7.3 Benchmarks

Also checks for fresh
ness specification if t
can for one be even a
plied to the system ,
due to ongoing con-
straints or if it is eve
possible to specify a
freshness index ¿1.0

Missing: **??**

Explain why it is nec
essary to verify the s

7.3.1 Evaluation Environment

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

7.3.2 Evaluation Procedure

The following steps outline the procedure for benchmarking data freshness within PolyphenyDB.

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 as different kind of

Talk about implemen tion of freshness cha teristic in many quer languages

Missing: Check if fre ness could even be used, or if we would always fallback to pr mary or when is the turining point when freshness now works.

Missing: Using 3 No and 5 Node with eager replication than same using lazy repli tion without any fres neess queries.

Missing: Locking switched from tablewise to partition-wis Compare unpartition table to horiontally partitioned table, sin gle placment vs. distributed across store

7.3.3 Results

The result generally shows

8
Conclusion

With this implementation Polypheny-DB now provides functionalities to adjust itself to the concepts revolving around *CAP* and *PACELC* described in 4.3. 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.

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.

8.1 Outlook

8.1.1 Tuneable Consistency

The introduced implementation sketched in section 6 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 8.2.1 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.

8.1.2 Locking

Reduce locking to a physical partition level (partition placement)

8.2 Global Replication Strategies

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

8.2.1 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 [22] 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.

8.2.2 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] 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.

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

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.

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

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

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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 7.

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 .