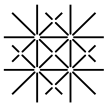

University of Basel

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 [**?**]. 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 [**?**] [**?**].

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 [**?**] [**?**]. 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[**?**]. 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.

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

1.1 Goal?

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. Foruth, 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 **??** motivates and illustrates the benefits of a freshness-aware data management approach and how it would impact a given scenario. In Chapter **??** 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 **??** illustrates existing fundamental concepts in the context of distributed data management systems. The system architecture of the polystore: Polypheny-DB is described in Chapter **??**. Chapter **??** describes the necessary 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 Polypheny-DB. The implementation of the freshness notion as well as coexisting building blocks are described in Chapter **??**. While Chapter **??** focuses on possibilities how to ensure correctness and measure the performance of an implementation, including all necessary prerequisites. Finally, Chapter **??** concludes the thesis by summarizing the individual contributions according to the proposed implementations and gives an outlook to future work and possible extensions.

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 .
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
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

This might not be su able in Motivational Scenario

Missing: Cite this

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.

2.1 Goal?

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

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 for the respective sections. It is separated into ... sections, which are characteristic for Data Freshness. The definition of Data Freshness ($\rightarrow$ *section*??)

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 (S2PL) 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 [**?**]. 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 [**?**]. *BASE* in this context stands for **B**asically **A**vailable **S**oft State and **E**ventual Consistency [**?**]. 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. [**?**] 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 how to characterize and measure data freshness exist. In 1999 Naumann et al. [**?**] 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 [**?**] 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.
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 [**?**] also described freshness as a matter of how old data is and linked this to an users expectation who is interested in 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. It 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 [**?**][**?**][**?**] 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 [**?**] and [**?**] define the notion of freshness by relating it to the age-of-information (AoI) respectiviely 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.

3.2.1 How to express freshness

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

- **Time-bound constraints**: Are used as timestamp that tolerate values which were updated by younger transactions.
- **Value-bound constraints**: Consider percentage deviation 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 [**?**]. 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 its timestamp **t** of the most recent committed update transaction that has updated 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. [**?**] 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 [**?**] and [**?**] 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.

3.3 Update propagation - Replication

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 [**?**] [**?**] [**?**] have pointed out in their work.

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

Wei et al. [**?**] propose a replication in form of an update policy adaptation algorithm that dynamically selects update policies for derived data items 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.

Psaroudakis et al. [**?**] 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, their focus here rather resides on a single table level than on a replication scenario in a distributed environement. 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 [**?**] which aimed to mitigate complexity and replication overhead by combining base data elements and derived elements. For queries a base set of information is enriched with a delta to get the relevant derived 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. 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 [**?**] 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 not updated frequently. 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, [**?**] 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. This as well corresponds to a multilayer replication setup.

3.3.1 Refresh strategies

Based on the discussed lazy propagation mechanisms in **??**, three refresh and update strategies could be identified when updates should be propagated to replicas.
Although the authors of [**?**] 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 [**?**] 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 not to high and is therefore able to endure such an update [**?**].
Contrary to [**?**] 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 of the users requirements. Any outdated node independently will pull the number of updates which is required to fulfil future requests of such requirements.
Wei et al. [**?**] 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.

3.4 Consistency

There are several constraints to consider that need to ensure the consistency in a distributed setup. In general two requirements can be defined that have to be fulfilled to guarantee correctness in respect of freshness. For one, transactions that refresh stores need to be executed on all read-only nodes in the same serialization order as the original update transaction. Secondly all queries of a read transaction must access data with the same freshness.
The authors of [**?**][**?**] 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 [**?**].
Voicu et al. [**?**] 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 [**?**] introduced a common data structure for both workloads for delta and main store and the usage of Muti 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. 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 [**?**] 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 beeing read within one transaction.

3.5 Freshness-aware Read Access

Since the essential idea of data freshness revolves around fast read access while transactional workload is currently beeing 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 [**?**] a router needs to utilize system state information on replicas to identify the correct way to route a query.
In [**?**] 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 posses a sufficent 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 [**?**] 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

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.

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 [**?**].
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 [**?**].

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* [**?**]. 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 [**?**].
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 [**?**].

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 [**?**]. This approach is often used in the context of normalization of a data model [**?**]. 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 [**?**]. 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 [**?**].

4.3 Temperature-aware Data Management

4.4 CAP Theorem

An essential part of distributed systems is the handling of failures or outages of participating components. The *CAP theorem* [**?**] introduced by E. Brewer [**?**] 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 [**?**] 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.

4.5 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 [**?**].
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 [**?**]. However, maintaining the consistency of all available replicas, results in scalability problems.
The author of [**?**] 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 **??**.
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 by 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 [**?**].
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 [**?**]. 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 [**?**] However, during

the convergence period, until all changes have been replicated, the system is exposed to an inconsistent state. As pointed out by [**?**] 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. [**?**]

4.5.1 Eager Replication

4.5.2 Lazy Replication

4.6 Concurrency Control

A major topic in the context of distributed database systems is *concurrency control* [**?**]. concurrency control [**?**]
As described in Section **??**, PACELC deals with the trade-off of latency and consistency in non-failure scenarios. In this context a major and also extensively studied problem of distributed database systems is concurrency control .
- Discuss downsides of SS2PL because eagerly replicated and not allowing much performance but ensuring high consistency.

Missing: Cite

Missing: COnvert to bullet points

4.6.1 (SS)2PL

4.6.2 MVCC

Established for multi-version databases that extend the concept of shadow pages by keeping the complete history or at least multiple versions of each object [**?**]. MVCC was introduced by [**?**]
Snapshot Isolation [**?**] snapshot isolation in lazy replicated systems is difficult
Snapshot Isolation provides weaker consistency guarantees as *one copy serializability* (1SR) and can therefore increase concurrency by relaxing the isolation requirements. Snapshot Isolation was introduced in by Berenson et al. [**?**]

4.6.3 Discussion

Although, there now exist some work that provides serializable versions of multi-version concurrency control [**?**].
There exists to make SI serializable as sumamrized by the authors of [**?**]

5
System Architecture

In this chapter we briefly describe and illustrate a simplified version of Polyphenys current architecture.
This extends the foundations laid out in Chapter **??** and sets them in context of the existing system model.

Missing: Polypheny multi-model teherfor tables are considered entities

5.1 Polypheny-DB

PolyDBMS [**?**]

Add PolyDBMS cite Love Marriage or Ma riage of convenience

Polypheny-DB is an Open-Source project[1] developed by the *Database and Information Systems* (DBIS) group of the University of Basel.

Polypheny-DB is a self-adaptive polystore that provides cost- and workload aware access to heterogeneous data[**?**].
Compared to other systems like *C-Store*[**?**] or *SAP HANA* [**?**], 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 *Cassandra*[2], *PostgreSQL*[3] and *MonetDB*[4]. 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.

[1] https://polypheny.org/
[2] https://cassandra.apache.org/
[3] https://www.postgresql.org/
[4] https://www.monetdb.org/

5.1.1 Data Placements

5.1.1.1 Vertical Placements

5.1.1.2 Data Placements

5.1.2 Query Routing

5.1.3 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.

6
Concept

We propose a concept for freshness-aware data management along with all necessary requirements to establish the concept.

This chapter is separated into several sections were each represents a necessary building block to provide the notion of data freshness in Polypheny-DB. These blocks are compared and discussed to the implementations presented in **??** to propose approaches how such techniques could be applied to polystore systems. Essentially we have to think about how we want to express freshness, find a suitable metric to measure it and provide users a possibility to formulate an acceptable level of freshness. Based on these fundamentals we need to consider how update transactions can be decoupled and deferred, how replicas will be refreshed while ensuring the consistency of the system and how to enable the routing to identify freshness levels to speed up read-only operations.
In this chapter we will describe the functional requirements needed to establish a form of freshness-aware data management within Polypheny-DB.

As described in chap
ter... we differentiate
freshness in x parts
This leads us to the
lowing tasks to addr
and find solutions

6.1 Functional Requirements

as described in Section **??**, since read-only queries typically benefit directly from data partitioning, they are suitable candidates to base our freshness awareness on. Therefore, we propose to define 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.

Missing: List what
building blocks are
necessary to compos
a freshness concept.
Check if needed as s
arate section

6.2 Locking

Missing: List what
building blocks are
necessary to compos
a freshness concept.
Check if needed as s
arate section

6.3 How to express freshness

As discussed in **??**, freshness can be expressed via several indications. Although we have described a number of domains only the time-bound constraints will be pursuit further.

While freshness extractions on value-based divergence is only really suitable for numerical values to calculate the percentage of deviations from the base item, time-based freshness can be applied to arbitrary types and can therefore be used in a more general notion. Such time-bound constraints allow to utilize timestamps as well as the usage of freshness indexes when referring to a desired level of freshness. Although a timestamp can be used more intuitively, an index is abstract and can therefore be generally applied without having to specify an absolute time dependency.
Hence, we propose that users can specify their tolerated level of freshness in a variety of ways. For one we want to provide the possibility to let users request their accepted level of freshness as an attachment to the query language. This can be achieved by extending the query functionality of Polypheny-DB. Such constraints can either be formulated by providing a timestamp to implicitly give an accepted lower bound on an absolute time. Additionally, time delays such as *"Last 30 minutes"* can be specified to request a relative freshness level. Finally, by directly requesting a desired level of freshness defined as a percentage value, meaning a value of 70% would tolerate data objects with a flexible accuracy of 30% outdated data corresponding to the time since the primary copy has been updated. However, internally all specifications are converted into freshness indexes since they are independent of any time window and can be constructed, compared and applied without any restrictive dependencies.

Moreover, to improve the user experience we consider to enrich the query tree paths 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.

6.4 Replication

Missing: Fil

The system can implement each of these cases in various ways; however, each implementation comes with a consistency/latency trade-off.

6.4.1 Replication States

As already discussed in the implementations in **??** it is crucial to define and assign roles to loosen the locking situation on nodes to minimize update times over all. The related work clearly separated 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 by the primary nodes. Therefore, these are the only nodes used for read-operations. However, we want to include primary nodes for reads as well and therefore propose the naming-convention in respect of the desired state as *up-to-date nodes* for primaries and *outdated-nodes* referring to any secondary node.
With the introduced changes to internal partitioning described in [**?**], we now have the

possibility to utilize the notion of *Partition Groups* to label stores with desired state respectively their roles. When deciding to use freshness on an object, administrators should be able to define how many replication tiers they want to have in a particular setup. The minimum requirement is always a complete primary up-to-date node, either as a full placement or composed out of several vertical partitions and one outdated node. However, to support a multi-tier approach as [**?**], outdated nodes can occur multiple times and can hold different levels of freshness with any number of vertically partitioned placements. Therefore, a suitable mechanism has to be introduced which enables the definition of replication orders and paths. This could then be used to intuitively have placements at different freshness levels resulting in tertiary adapters only being updated by secondaries.

Missing: asyncronou immediate, adhoc, lo aware

6.4.2 Replication Strategies

Since data freshness occurs as a side effect of using lazy replication and can therefore already be used as a consequence of saving workload when not needing to update every replica eagerly. As suggested in **??** there exist several techniques how outdated nodes can be updated lazyily 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 the control and flow of data. Although in the proposed works some systems allowed to access read-only copies directly with Polypheny-DB we always have a single point of entry which can vaguely be compared as the poly-layer acting as a master node when distributing or even routing queries. Hence Polypheny acts as the master anyway. However since any requests have to pass through the poly-layer we have full control how and where queries need to be routed to allowing us to selectively route read-operations to outdated and up-to-date stores alike. This enables Polypheny-DB to take full control how different levels of data freshness are being accessed.
With this abstraction layer on top of the stores we can leverage the DBMS to act as a coordination service allowing us to loosen the eager replications and locking mechanisms to be restricted to the primary nodes only. Currently, one has to wait until an update is persisted everywhere and are therefore dependened on slowest performing store. To mitigate this behaviour we want to decouple the commonly used update transactions and logically divide it into a total of three separate transactions types to allow deferred refreshments of objects.

- **Update transaction**: Consequently are write operations that are targeted on primary placements only.
- **Propagation transaction**: Belong to a refresh operation updating the data freshness of an outdated secondary node which is executed automatically by the system.
- **Refresh transaction**: A propagation transaction which is manually triggered by a user to refresh the data on an arbitrary outdated node.

Allthough, logically being used differently they are technically executed with the same capabilities and only really separate in terms of locking. They are rather used as a clear

indication on which part of the process is referred.
These changes will also have an impact on the partition distribution constraints. For vertically and horizontally partitioned entities this means that all constraints on the number of available column-placements can only be considered for primary update stores. Otherwise, this could harm the consistency of the system. For nodes labeled as outdated any variation is possible.

6.4.3 Update Propagation

We have several possibilities how to handle and propagate the updates. Since every write-operation needs to go through Polypheny, we can easily keep track which operations have been applied to the primary server and when they need to be applied to the secondaries. Essentially every possible solution needs some form of queue to store updates that still need to be executed on secondaries. For correct serializability it is sufficient to store the commit timestamp of the update transaction with each queue item. This imposes a natural execution order of any item in the queue to be delivered to the secondaries. We propose the following approaches:

CDC (Change Data Capture)
We can implement the notion of a shadow table which is filled by trigger on the base table. As soon as the base table gets modified by an update transaction any changes applied are stored and consequently tracked. Accompanied by a commit-timestamp its content is then used inside propagation transactions to refresh outdated placements. Once placements are updated the propagated entries inside the shadow table are deleted. This has the advantage that we can utilize a common datastructure in Polypheny out-of-the-box without having to implement any further activities. It is persisted and available after a downtime which ensures the recoverability of the system. However since this results is generating several statements that essentially has to traverse the entire system in order to be executed this could lead to large overhead in the system which could impact the performance as well as requiring more resources. Furthermore, since this is an updatable table the implemented locking mechanisms has to be applied as well. This excessive accessing however could result in high locking situations and therefore again impact the update time after all.

Internal structure
Another possibility would be to handle this approach entirely with internal structures. For example with the implementation of a MOM (Message Oriented Middleware) between the poly-layer and available adapters. The initial update transactions can append the changes into a persistent queue. This queue can then be asynchronously queried using a pull-based approach on adapter site to apply all changes necessary for them.

Internal Partitions
Another propositions would be the usage of background processing together with physical partitions. Although, they were originally intended to serve the horizontal partition

use case they are internally configured to represent a physical table. Partitions could be extended to be used as general system internals which are accompanied by an update-timestamp that represents the last time the partition has been updated, this could include the Age-of-Information as well as the Age-of-Synchronization. Furthermore, since partitions logically belong to partition groups described in **??** we can easily identify which partitions are considered to be outdated or should always be up-to-date. With these information we could even use the existing capabilities of the *Partition Manager* which uses the notion of partition-qualifiers to identify required levels of freshness.
For every update-transaction the eager replication is loosend to only consider the primary partitions for updates. For the refresh operation itself we can utilize the *Data Migrator* to extract the relevant data via an accumulated select-statement on the primary placements. Additionally, to further decouple the data refreshment a hidden shadow partition can be created for this purpose. In that way writes on the primary copy as well as the currently existing outdated placements can still be executed while the outdated placement gets refreshed in the background. Once the data migration process has finished, we only need to logically apply a *Refresh-Lock* to ensure the overall consistency. During the short time the outdated-site is locked the old placement is dropped and the hidden shadow partition is officially being activated. Since this only really requires one select on the primary site, write-operations can still take place, reducing locks to a minimum. Even during the time the refresh-lock is active the benefits of the poly-layer comes in to play which enables the system to choose to route read-operations to primary servers as well.

View Materialization:
Along the idea of internal partitions materialization could help reduce the number of statements necessary to create new levels of freshness. Since materialized views are considered to be snapshots of data objects, anytime a propagation or refresh operation is being executed a new materialized view is generated on basis of the up-to-date placement. This would omit replication operations entirely hence no bookkeeping of the queued updates would be necessary. The only needed reference would be the new timestamp to calculate the freshness index.
As with the handling of internal partitions this can also be done entirely in the background. This limits the locking time to a minimum and is therefore only really necessary when switching the outdated with a refreshed view. Although, one of the biggest downsides of materialization is the large additional resource consumption on storage, it can be neglected for polystore systems, since they are inherently distributed and would otherwise store the entire content of the table. Therefore, it would even slightly reduce the data footprint on outdated placements.

6.4.4 Refresh strategies

As already stated above, there are essentially three ways to update a secondary node. For the automatic execution of propagation transactions a periodic execution has several downsides hence we propose to schedule refreshment on the basis of the load on the outdated

placements. Therefore, we will extend the adapter of the underlying stores to gather metrics on current workload average response times in order to enable the poly-layer to decide based on this metrics when it is suitable to carry out a propagation-transaction.
Since it might be possible that a user may consider refreshing a placement manually, we have proposed refresh transactions. With this a user has the possibility to refresh any replicas which are classified as outdated to a specific freshness level.

```
ALTER TABLE dummy REFRESH PLACEMENT ON STORE outdated_store
UNTIL <TIMESTAMP>;
```

Without any specification the placements shall be updated to the most recent state of the up-to-date version:

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

For this operation only freshness levels greater than the current local freshness level can be considered. Furthermore, this should also provide the possibility to refresh all placements of an object.

```
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.

6.4.5 Identify stale data

A prerequisite for the automatic execution of propagtion transactions is the identification of stale and outdated placements. Although a trivial option would be to use the roles described in **??** and periodically gather all labeled placements, this would not scale much considering a large amount of tables with at least one additional replica per table.

One proposition would be after every update transaction has commited to immediately inform dedicated notification services which will instantly trigger a load analysis on stores to be replicated. A sligthly different approach would be a central gossiping service as an adapter extension that gathers store metrics and declares them as idle, or loaded. With such a mechanism the notification service which received the information to update stores would not start querying a server again and can directly utilize the pre-gathered metrics and choose to execute the update or defer it again.

Another approach to identify and update stale placements would be the idea to use update-on-read. Everytime a node flaged as outdated is used to serve a read-operation the level of freshness can be checked. If identified as outdated this could then trigger any of the propagation transactions proposed in **??**. This would completely omit periodic background scheduling and would shorten processing time.

However, for both cases in terms of the resulting refresh rate we still have to consider if the change data becomes too large the refreshes tend to take longer. So countermeasures have to be defined to minimize errors or downtimes.

6.5 Transactional guarantees - Locking - Isolation Level

To support the overall consistency of the system we have to ensure that all updates are correctly serialized and applied to the system.
Polypheny for one ensures global atomicty by using the two-phase commit (2PC) protocol and for correct isolation treatment strong strict to phase locking (SS2PL). Currently, all placements are eagerly updated and locked whenever a write-operation is executed. We need to loosen that constraints to increase update times and still ensure all transactional guarantees. The locking mechanism should therefore now only consider primary nodes. In that way we still can utilize secondaries with slightly outdated data for reads.

To ensure that the overall correctness of the system is maintained we will restrict the solution to allow freshness related parameters solely for read-only transactions. For every update-transaction which considers freshness of any kind, consistency cannot be guaranteed since we might read outdated data and use this data to write into another table. Therefore, we propose to enrich the transactional context of Polypheny with a flag that states if any read-operation considers freshness. If this is the case, the update-transactions needs to be aborted.

Another important point to note is the referential integrity of outdated tables. We already ensured that outdated data cannot be used within update-transactions to ensure consistency and therefore guarantee correctness. However, for read-only transactions it should still be possible to join any tables while considering a desired level of freshness. Since it is rather complex and would result in refactoring the core of Polypheny-DBs, we currently refrain from supporting MVCC. Hence, it might occur that cumulative reads on multiple objects might return incomplete results, since a specified freshness level can be entirely different on two tables which makes it hard to even compare freshness levels among different objects. However since a user is willing to access stale data anyway this is a known risk and thus can be accepted.

Finally, as introduced in **??** the *Refresh-Lock* is a newly introduced lock which holds the same characteristics as a regular lock on an object. However, it shall only be applied whenever a refresh operation is currently in place. This way the routing mechanism can avoid sending any queries to that placement for reads or a new refresh operation, which might have been triggered manually by an user.
To ensure the correct execution of refresh operations every placement that is outdated will receive an independent propagation- or refresh-transaction. The serialization order of the updates to be executed is the commit order of the initial write transaction. These can also be refreshed and updated independently which not only again 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.

6.6 Freshness-aware Read Access

With the description of **??** how to explicitly express freshness, users have the ability to hint or even guide the routing process to select placements based on their desired freshness.
Users can choose to specify any tolerated level of freshness. The routing process analyzes this specification and gathers all placements which fulfil the necessary requirements by preferring secondary nodes labeled as outdated. In case no secondary node can fulfil the request a primary placement is used to serve the query since they are not exclusive to receive updates such in other systems, consequently utilizing all source available to the system. Since only few systems presented in **??** provided this functionality the routing process can be extended to support load balancing. With the proposed adapter background analysis the router can observe if any selected placement might be overloaded and therefore chooses to route queries to a different available location.

For the overall freshness guidance an extension of PolySQL is necessary. Along the description in **??** users can choose to select any of the specifications to guide the system.

```
SELECT * FROM dummy
[ WITH FRESHNESS [ <TIMESTAMP> | <DELAY> | <PERCENTAGE> ] ];
```

As an extension we also propose to omit the specifications entirely and only state to the system that any freshness level is acceptable or the system central configured default, to further speed up queries.
As mentioned in **??** since update transactions don't allow any usage of freshness metrics to ensure consistency, solely read-only transactions are allowed. Hence, for read-operations on outdated nodes and data, locks can be omitted entirely since you will read stale data anyway. We therefore only have to validate that no refresh-lock is currently in place.
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 adminsitrator 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 [**?**] 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 storages perfromance 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.

6.7 Freshness Metrics

As described in section **??**, there is no unified definition of data freshness or common freshness metrics. These rather depend on specific use cases and system requirements.

6.7.1 Time-based constraints

6.7.1.1 Absolute Timestamp

6.7.1.2 Absolute-Delay - From Current Time

Both reconstruct a timestamp that enacts as a lower bound of acceptable placements. This can be used as a filter to check for each candidate placement that it is fresher or respectively has received a state where its commit time is newer than the lower bound. If not the placement is removed from the possibly candidates.

6.7.2 Replica-deviations

6.7.2.1 Relative-Delay - Deviation From Eager Replica

Although also based on a time difference it checks relative to the specified time delay on teh eager replica how far away is the lazy replcia.

6.7.3 Data Accuracy - Modification - Deviation From Eager Replica

This can also be used in referential integrity when joining multiple table entities. We can use the joint number of modifications and compare it against the current number of update transactions to give a joint accumulation.

6.7.3.1 Data Accuracy - Time - Deviation From Eager Replica

Both construct an index or percentage of "hit ratio"

6.7.3.2 Access-Update Ratio - AUR

6.8 Consistency

Move from Strong consistency to eventual consistency have a loose form of consistency constraints that we need to comply with, in order to still provide response times while sacrificing consistency.
This is certainly we still have to comply with the acid properties.

Missing: Introduce c straints on freshness and DML operations within the same tran action

Missing: Add this to requirements when talking about what is needed to provide freshness-aware data management

6.9 Placement constraints

7 Implementation

This chapter describes the implementation of .. . The component and modules of these systems are described in Chapter **??**

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

7.1 Locking - Isolation Level

7.2 Refresh Operation

7.2.1 Replication Strategy

Can be used to selectiviely

7.2.2 Replication State

UPTODATE

REFRESHABLE

INFINITELY-OUTDATED

7.2.3 Replication Algorithm

7.2.4 Manual Refresh Operations

7.3 Freshness Query

7.3.1 Freshness Query Specification

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

7.3.2 Freshness Filtering

7.3.3 Freshness Selection

Missing: Since the replication or refresh propagation is done operation-wise we ac ally loosened the hea SS2PL constraints th we require on the pr mary updates. Since we already have a se alizable schedule afte execution we also kn per entity and each p tition placement the correct execution orc that we have to appl the data changes on. it is not necessary to free the resources aft the entire transactio has been replicated

7.3.4 Referential Integrity

8 Evaluation

This chapter is separated into

8.1 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

8.2 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.

8.3 Benchmarks

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

8.3.1 Evaluation Environment

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

8.3.2 Evaluation Procedure

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

8.3.3 Results

Execute benchmarks multimodal dbs. as as different kind of

8.4 Discussion

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

The result generally shows

9 Conclusion

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

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 Serialiazable 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.

9.1 Outlook

9.1.1 Tuneable Consistency

The introduced implementation sketched in section **??** 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 **??** 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.

9.1.2 Locking

Reduce locking to a physical partition level (partition placement)

9.2 Unified transaction model for semantically rich operations

Introduce semantically rich operations inside polypheny? Maybe not on the scope of an application level but to encompass the underlying transactions

9.3 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.

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

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

This chapter includes all utilized query templates for evaluating freshness-aware data management within Polypheny-DB.

C
Evaluation Results

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

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 .