Optimizing Consistency and Performance Trade-off in Distributed Log-Structured Merge-Tree-based Key-Value Stores

by

Viraj Deven Thakkar

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved June 2023 by the
Graduate Supervisory Committee:

Zhichao Cao, Co-chair
Xusheng Xiao, Co-chair
Chris Bryan

ARIZONA STATE UNIVERSITY

August 2023

ABSTRACT

Distributed databases, such as Log-Structured Merge-Tree Key-Value Stores (LSM-KVS), are widely used in modern infrastructure. One of the primary challenges in these databases is ensuring consistency, meaning that all nodes have the same view of data at any given time. However, maintaining consistency requires a trade-off: the stronger the consistency, the more resources are necessary to replicate data across replicas, which decreases database performance. Addressing this trade-off poses two challenges: first, developing and managing multiple consistency levels within a single system, and second, assigning consistency levels to effectively balance the consistency-performance trade-off.

This thesis introduces Self-configuring Consistency In Distributed LSM-KVS (SCID), a service that leverages unique properties of LSM KVS properties to manage consistency levels and automates level assignment with ML. To address the first challenge, SCID combines Dynamic read-only instances and Logical KV-based partitions to enable on-demand updates of read-only instances and facilitate the logical separation of groups of key-value pairs. SCID uses logical partitions as consistency levels and on-demand updates in dynamic read-only instances to allow for multiple consistency levels. To address the second challenge, the thesis presents an ML-based solution, SCID-ML to manage consistency-performance trade-off with better effectiveness. We evaluate SCID and find it to improve the write throughput up to 50% and achieve 62% accuracy for consistency-level predictions.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Distributed Key-Value stores, specifically Log-Structured Merge-Tree-based Key-Value Stores (LSM-KVS), play a vital role in modern-day infrastructure. LSM-KVS are extensively utilized in various applications, such as the highly reliable etcd [7], Kubernetes [10], and the highly scalable ArkDB [38], as well as in large companies like Meta [35, 36] and Alibaba [14, 38]. However, ensuring consistency across replicas remains a challenge for these distributed LSM-KVS. Here, database consistency refers to the guarantee that all system nodes have an identical view of the data, regardless of the timing of updates.

To address the consistency challenge, distributed LSM-KVS must navigate the trade-off between consistency (C), availability (A), and partition tolerance (P) as outlined by the CAP theorem [28]. The trade-off arises due to the inevitability of network failures, requiring tolerance of network partitioning. Consequently, database engineers need to choose between ensuring consistency or availability since these two properties are inversely related by definition. Strong consistency, where a write must be propagated to every node before any new reads/writes can be performed, reduces availability, where a database should be available for writes at all times. Conversely, the high availability of a database implies weak consistency, where data may be provided to users with outdated information until eventual replication occurs.

To effectively manage these trade-offs and provide flexibility, alternatives like multi-consistent databases have been developed [41]. These databases enable multiple consistency levels within a single distributed database, allowing users to choose the appropriate level for different portions of data and achieve flexible choices ranging

from strong to eventual consistency within the same database. However, designing such a database and ensuring high performance pose significant challenges. Current implementations [6] of multi-consistency still suffer from a major difficulty: when data is written with strong consistency, the database temporarily experiences low availability across the entire system until the data is propagated to all nodes. This behavior is undesirable and highlights the challenge of providing high availability for parts of the database not associated with strong consistency.

Creating a database with multiple consistency levels is challenging, as is managing the balance between consistency and performance. Incorrectly identifying strongly consistent data as weakly consistent can result in losing data correctness at replica nodes. On the other hand, misidentifying weakly consistent data as strongly consistent can lead to a loss of high availability. Therefore, it is crucial to accurately identify consistency levels to maintain optimal database performance. The responsibility of managing the trade-off between consistency and performance lies with the application developers, as each project requires a different balance. When a database is shared by multiple teams, developers must correctly assess the consistency level of their data relative to that utilized by other teams. Incorrect identification of consistency levels can lead to a loss of correctness or availability.

This thesis identifies two major challenges in designing and optimizing multi-consistency distributed LSM-KVS. Firstly, creating and managing multiple consistency levels within the same distributed LSM-KVS is difficult [41, 39], requiring solutions for data synchronization, replication, and request forwarding issues. Secondly, optimizing the overall performance of LSM-KVS with multiple consistency levels and reducing the burden of manual consistency level choices for developers is an unexplored issue. To address these challenges, this thesis presents Self-configuring Consistency In Distributed LSM-KVS (SCID), an ML-based solution that manages

the consistency-performance trade-off in a distributed, scalable, and nodal LSM-KVS.

To overcome the first challenge, SCID leverages the unique properties of LSM-KVS, including logical KV-pair-based partitions and low overhead of dynamic read-only instances without data replication. LSM-KVS exhibits high write performance [1, 40]. To maintain strong consistency, every write operation must be propagated to all nodes, which can significantly impact write performance. To mitigate the availability impact of strong consistency, SCID maintains only one KVS instance for a specific key range instead of multiple replicas. This approach saves space, avoids data synchronization during writes, and assumes the reliability and availability of the data stored in disaggregated storage (e.g., HDFS or S3) [9, 5].

Secondly, to support intensive read requests, SCID utilizes read-only instances to respond to queries with different consistency levels. The read-only instances can access all data from the primary instance, ensuring eventual consistency.‘ the read-only instances can serve reads for data not yet flushed to storage by the primary instance through online synchronization initiated by the read-only instance itself. Dynamic read-only instances, capable of updating data on-demand, enable retrieval of the latest data from the primary instance. Furthermore, SCID leverages logical KV-pair-based partitions to manage data with different consistency levels, allowing each partition to have an individual consistency level. The dynamic read-only instances update the data in each partition through on-demand calls. These design choices enable SCID to achieve multiple consistency levels without data replication and synchronization during the write path, ensuring optimal write performance and space efficiency. The read-only instances cater to intensive read operations, and their number can be adjusted based on the workload.

Currently, developers are typically responsible for assigning consistency levels to each KV-pair in databases [13, 42, 12]. However, as applications and product scales

increase, and database centralization becomes necessary, this task becomes challenging. It requires considering the future state of the application and the trade-offs with other teams' data. Incorrect estimation of consistency levels can lead to over- or under-estimation. To tackle this challenge, accurate consistency-level assignments and predictions are necessary. These predictions should be resilient to spontaneous situations with complex patterns that may be too intricate for simple rule-based solutions. ML models excel at discovering relationships between data attributes and appropriate consistency levels. SCID leverages the predictive capabilities of ML models.

In scenarios like Key-Value stores where only two columns are present, traditional solutions using pre-analyzed data attributes and organizational requirements are insufficient. To address this limitation, we propose using ML models specifically tailored for Key-Value stores, testing various models, and selecting the random forest model based on its accuracy and low latency. Additionally, we propose predicting consistency levels at the read path, enabling the query to be redirected to either the primary instance or one of the read-only instances (which may have different consistency levels based on their synchronization status). The returned KV-pair contains the consistency level tag. In case of incorrect consistency level assignment, the application can resend the request with explicit consistency requirements to the shard, incurring extra overhead. Therefore, a high prediction accuracy of consistency levels is crucial.

We implemented SCID based on RocksDB and utilized Kubernetes and Docker to manage the read-only instances that share the data. In the evaluation, we divided the entire dataset into multiple shards based on the key range. Each shard has only one primary instance responsible for write requests and most of the strongly consistent read requests. The read-only instances of each shard are dynamically launched or shut

down by Kubernetes based on workload. We used YCSB [23], a widely-used NoSQL benchmark, to evaluate SCID. Compared to Master-Slave-based distributed LSM-KVS designs, SCID achieved up to 50% better write throughput while maintaining consistent performance across varying consistency distributions. We also evaluated the ML model used, comparing it against a random selection model due to the absence of existing prediction solutions. The proposed SCID-ML model outperformed the random-selection model by 21%.

The structure of this thesis is as follows. Chapter 2 provides background information on relevant domains, describing the procedures involved. Chapter 3 presents the motivation and challenges of maintaining multi-consistency levels in distributed LSM-KVS. The design details of SCID are outlined in Chapter 4, while Chapter 5 delves into the implementation details. Chapter 6 evaluates and presents the results of SCID.

Chapter 2

BACKGROUND AND RELATED WORK

This chapter provides the technical background and discusses related work in the field of LSM-KVS (Log-Structured Merge Key-Value Stores) and database consistency in distributed setups.

2.1 LSM-based Key-Value Stores

Key-value stores (KVS) are simple associative array databases where a string-type key uniquely identifies the string-type value. This simplicity is a huge factor in the high performance of the datasets and the rising popularity of KVS. To construct the high-performing and fault-tolerant KVS, a variety of data structures such as B+ Trees, Radix, Hashtables, and Log-Structured Merge-tree (LSM-Tree) have been implemented. Among them, the LSM-tree structure is one of the most popular and widely used designs in mainstream KVS today [2, 1].

LSM trees have an append-only implementation, allowing for efficient writes by storing data in memory and persistently in immutable Write-Ahead-Log (WAL) files on disk. This implementation gives them high write throughputs and good read performance compared to other read-optimized KVS. These properties lead to LSM-based KVS like Google's LevelDB [27], Facebook's RocksDB [26], and DGraph's BadgerDB [3].

This thesis leverages some unique properties that LSM-based KVS as logical partitions [15, 4] and dynamic read-only instances [11] to improve read performance. The conjunction of these properties is present in Fig. 2.1. The dynamic read-only instances in LSM-KVS can update or be initialized directly from the storage com-

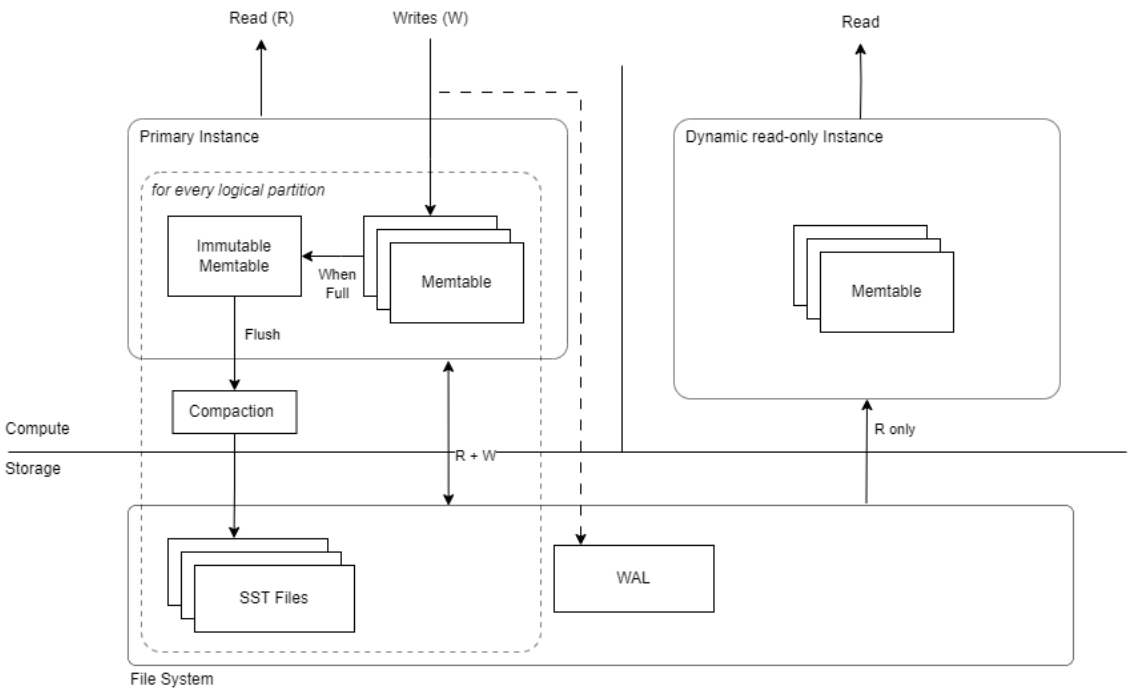


Figure 2.1: Process of Writes in LSM KVS

ponent using the WAL and SST files. Offloading Reads to the secondary instances makes the primary database instance available for Writes, improving throughput.

Writes. For every write, LSM-KVS will write data to 2 places, its in-memory MemTable (the write cache) and on-disk Write-Ahead Log (WAL). Once the memtable reaches a specific size limit (user-configured or default) or a manual Flush is called, the memtable and WAL become immutable, and a new memtable and WAL are spawned for new writes. The contents of the memtable are then flushed into the storage system as an SST file on disk following which, the old memtable and WAL will be deleted. The behavior can be identified in popular databases like Cassandra and RocksDB.

Logical KV-based Partitions. LSM-KVS uses WAL and SSTs primarily to store data, and after every Flush, the data is persisted in the SST files. However, this

process can be modified in the middle, where while the database shares the WAL, there is a logical partition between the SSTs. This logical partition each has its properties and uses. This has already been implemented in popular databases with the term 'Column Families' synonymous with it in RocksDB, Cassandra, HBase, and BigTable [45, 21, 26]. While the implementation differs in every database, the core principle remains the same to achieve logical data separation in the same database.

Dynamic Read-only Instances. Much like read-only instances, these instances allow for read operations to the database without any ability to edit the database contents in any way. Read-only instances read the on-storage data of the primary instance and use the "on-storage snapshot" to serve the reads. Therefore, it can be quickly launched and respond the user requests. Dynamic read-only instances are much like read-only instances, with the additional ability to update themselves with new data with a special `OnDemandSync()` function. This function pulls data from the WAL (data that is cached in the primary instance's memtable) and the SST files. Since data is read directly from the disk, it has minimal overhead and allows for the dynamic instance to be more efficient than traditional read-only instances. Such a feature has been implemented in RocksDB and referred to as "Secondary Instances" [11].

2.1.1 Distributed KV-Stores

Single Instance KV-Stores are usually libraries like RocksDB [26] and LevelDB [27] and need to be integrated into applications. These embeddable key-value stores lack the scalability expected for modern databases and need to be modified for use as Distributed DBs. Facebook's ZippyDB [35], Apache's HBase [45], and Google's BigTable [21] are examples of such distributed databases. These databases are highly reliable and exhibit the properties of the CAP theorem as expected from distributed

databases. All these designs replicate the database to multiple replicas, which can achieve better fault tolerance against both hardware and software issues. This also leads to direct applications, like Netflix's EVCache [18] uses KV-stores as their caching solution for their most-used content, LinkedIn's FollowFeed [8] uses KV-Stores to organize the timeline for their platform's posts.

These Distributed KVS are often provided as cloud services, Google's BigTable being the prime example. And while data storage costs are low, they are substantial when considering the scale of a lot of applications like YouTube, Instagram, or Spotify, and having a copy of all data at every single node in a distributed system is not ideal [34].

Existing distributed LSM-KVS designs have two overheads:

- **Storage Overhead** as having a copy of all data at all locations is not ideal.
- **Network Overhead**, especially in times of peak performance when any additional overhead would greatly affect performance.

2.2 Database Consistency

Every Distributed database has to ensure that data read at different nodes is the same, this synchronization of data is irrespective of time. If done on the very write that data comes in, it is known as strong consistency, and if done at a later time, this is known as eventually consistent. What data to replicate is ensured with consensus algorithms (PAXOS [31], Raft [37] or variations of the same) or non-consensus algorithms (Last Write Wins, Conflict-free Replicated Data Types (CRDT) [43]). However, when to replicate is decided by Consistency levels [19].

2.2.1 Multi-Consistency Databases

Every application needs a different level to suit its use case and some applications even serve multiple use cases needing multiple databases. This requirement has led to the creation of various consistency levels, most prominently Casual consistency, and also led to the creation of database systems having multiple consistency levels. Prominent modern databases like Cassandra [6], MongoDB [41], and ZippyDB [35] all have implementations of this multi-consistency model.

Such a multi-consistency model is helpful for applications as every application has a different requirement for consistency, and often, the same application can also require multiple consistency levels. An example of this would be posts on a social platform can have eventual consistency and direct messages on the same social platforms need strong consistency. And instead of choosing to have a second database which increases costs, maintenance, and workforce; the multi-consistency databases can be easily configured to manage for strong and eventual consistency. However, it should be noted that these multiple consistency levels are provided based on the number of replications being made, essentially making it an extension of weaker consistency models.

2.2.2 Automating Consistency Classification

Having multiple consistency levels also means that there is a need to assign data to one of these levels. The current methodology is for the application developers to manually assign the key-values pairs to particular consistency levels which has the downside of the developers misidentifying the consistency level required by over/underestimating the relative importance of their data compared to other data.

Researchers have since taken this topic up to help determine how data should be

classified to different consistency levels. RedBlue [33] consistency is an early example of this, where the researchers propose a set of conditions to be met to determine what data is supposed to be strongly consistent (red) and what is to be eventual consistent (blue). This concept is then extended by the work in SIEVE [32] where the researchers automate this data distribution into either strong or eventual consistency using static and dynamic analysis of data. This analysis is done using pre-conditions on the data and the methodology limited to a per-use-case basis.

Researchers extend this in Indigo[17] by performing post-condition static analysis to prevent invariant violation for concurrent operations. And work done in QUELEA [44] allows for fine-grained application-level consistency properties using contract enforcement systems to generate an appropriate consistency protocol. However, all these methods need an extensive amount of knowledge that is application specific which is where the more recent work of AUTOGR [46] furthers the field.

Researchers in AUTOGR release this burden from programmers by not requiring them to specify application-specific invariants. The approach taken now requires zero human intervention and only needs the application codes as input. The primary algorithm used by them (RIGI) infers path conditions that could lead to consistency violations and avoiding them. Their research differs by focusing on a single consistency level and achieving geo-replication with that instead of having developers manually define consistency levels required by particular data.

All this research [33, 32, 17, 44, 46] has been performed on column-rich databases, where the value from the columns make it easier to achieve classification, and not much attention has been given to Key-Value stores.

Chapter 3

MOTIVATION AND CHALLENGES

Modern distributed LSM-KVS serve multiple purposes. Common examples like ZippyDB are known to store file-system metadata (usually strong consistency) to keep event counters (usually eventual consistency) [35, 16, 20]. This leads to an inherent existence of consistency gradients, ranging from hot to cold data. Every point in this gradient has a different performance trade-off, strong consistency trading off performance much more than eventual consistency.

3.1 Motivation

When considering databases with a single use case, this consistency-performance trade-off can be handled easily due to the pre-defined consistency level and performance expectation. However, in modern databases supporting multiple consistency levels, losing any performance leads to slowdowns for all data, including one that is a higher priority, and the better possible performance-consistency trade-off is needed. For this optimum, data needs to be assigned to the correct consistency level which currently is the responsibility of developers. And while this step allows for granularity and flexibility, it also introduces the likelihood of misidentification of consistency levels when considering companies at scale with multiple teams working on multiple products.

To avert these situations, every aspect of all the products should be known before any team can decide on what consistency level would be required for their product. Furthermore, this solution lacks flexibility beyond a limited number of consistency levels. For instance, it is more feasible for a team to assess the importance of their data

on a scale of 1 to 5 rather than on a scale of 1 to 100. Such an option is expensive both in terms of time and monetary value, raising the question if something better could be possible. Further, current implementations of deciding when consistency is met are based on the number of nodes where data is replicated, rather than consistency levels where the machine inherently should have the ability to decide. This is an important factor to consider since allowing the machine to decide on consistency syncing has the potential to make the process of writes much faster. Therefore, designing and developing a distributed LSM-KVS with multiple consistency level support and achieving a better tradeoff between performance and consistency level guarantees is essential but challenging.

3.2 Challenges

Solutions for such a problem can reside in the automation of this process, right from the selection for an accurate consistency level to allowing the database service to decide when the data is to be synced up. Current work for performing consistency-level selection exist and uses methods varying from rule-based approaches to more complex ML-based algorithms. However, work done is scarce and for column-rich databases [32, 17, 44, 46, 22]. This thesis focuses on Key-Value based data, attempting to resolve the two questions above in KVS.

To resolve these questions for distributed LSM-KVS, we propose the solution which is a prediction algorithm to determine what the consistency level for each key-value pair is supposed to be. This prediction algorithm is responsible for maintaining the performance-consistency balance and a prominent challenge is to ensure that the data is identified correctly. Additionally, scenarios when data is misidentified need to be devised to make sure that leash performance is lost and the fastest resolution is possible. This even leads to ensuring the availability of the database is not being

hampered in case there are misidentifications.

Further, another major challenge is to devise a consistency syncing mechanism that allows for the data synchronization mechanism to be handled at the replicas when data is requested rather than when written. This setup needs a combination of both, disjoint and selective synchronization of data. A possible solution for such synchronizations is the combination of the properties of dynamic read-only instances and logical kv-based partitions in LSM-KVS. Such a setup only being possible due to how data is only persisted in LSM-KVS on performing compactions and how dynamic read-only instances fetch data. The primary contributions of the thesis are:

- Reducing the Data Overhead: A system implementing disaggregated persistent storage. Instead of having data stored at all instances, a highly reliable and available persistent data storage is utilized.

- Reducing the Performance Overhead: On writes, data needs to be replicated to multiple instances in case of stronger consistency, this thesis implements a system that avoids this performance and network overhead.

- Reducing the Programming Overhead: Having the developers assign data may lead to misidentification for applications at scale. To simplify this process, this thesis proposes an automated consistency assignment system for KVS.

Chapter 4

SYSTEM DESIGN

In this chapter, we introduce SCID, a distributed LSM-KVS that achieves a superior balance between consistency-level guarantees and performance. Unlike traditional approaches that rely on manual consistency level selections, SCID incorporates automatic consistency level selection in the read path, reducing development overhead.

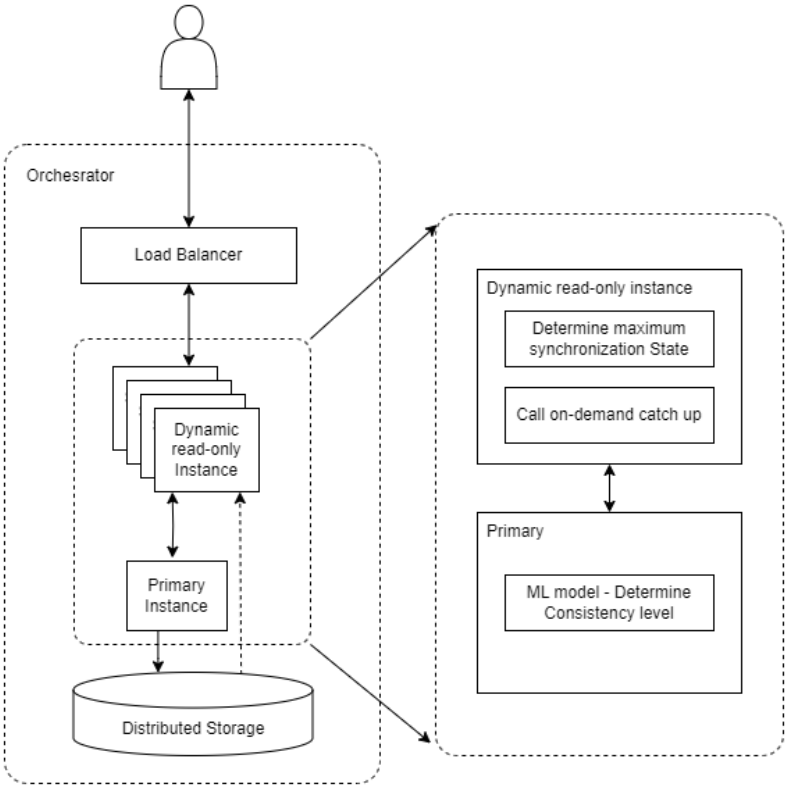
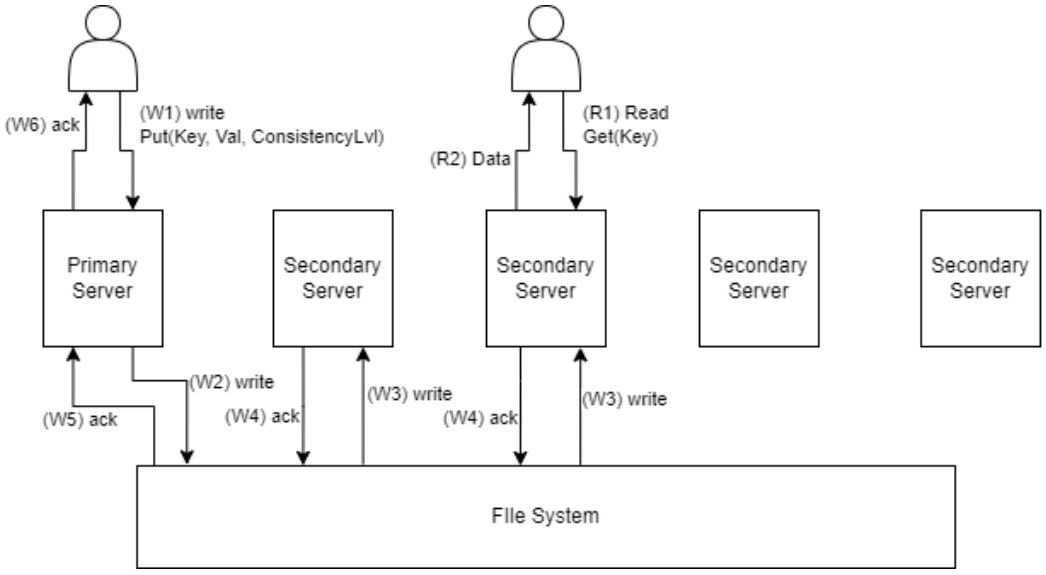


Figure 4.1: SCID Architecture

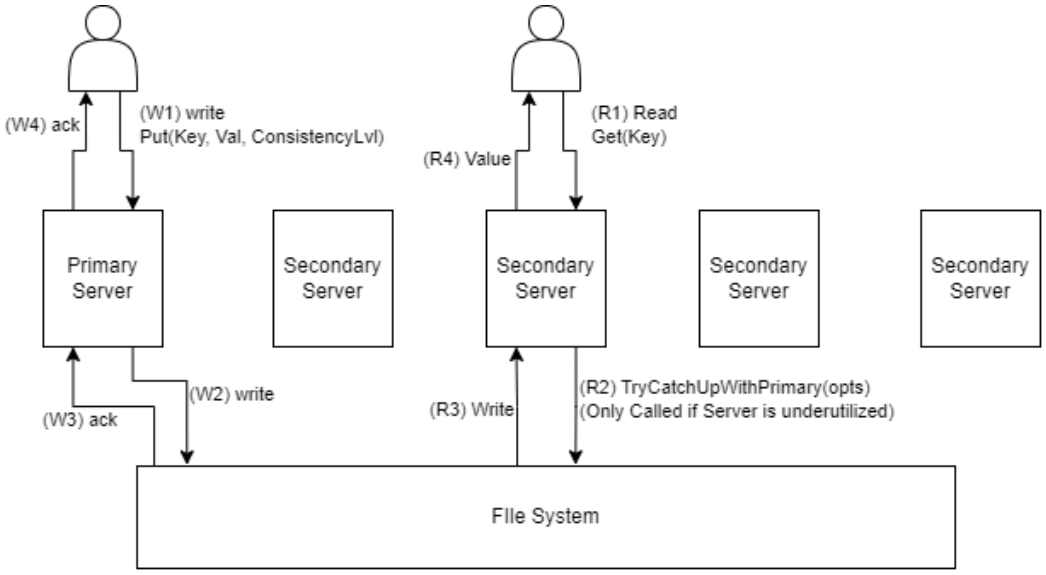
4.1 Tunable Consistency in Distributed KVS

SCID builds upon the strengths of LSM-KVS while introducing changes to enhance the tradeoff between consistency-level guarantees and system performance. SCID's architecture, depicted in Fig 4.1, assumes deployment on a storage-compute disaggregated infrastructure. The storage layer provides services over the network (e.g., HDFS or S3) and ensures data availability and reliability. In traditional distributed LSM-KVS systems, multiple replicas are used for certain key ranges to provide consistency guarantees. Immediate reads return either the new or old value, depending on the replica accessed. However, SCID takes a different approach by moving the consistency guarantee operations from the write path to the read path. In SCID, writes are only applied to the primary instance, returning immediately without data synchronization with the replicas. Figure 4.2b illustrates this approach, where strong consistency levels lead to immediate selective replication and consistency, while weaker levels return older data. This design is based on the assumption that non-immediate reads are likely to have the latest data due to periodic replication regardless of consistency methodology.

To enable selective replication of data, SCID combines the properties of Logical KV-based Partitions (Column Families) and Dynamic Read-only Instances. Dynamic Read-only Instances support a call called `OnDemandSync()`, enabling a read-only instance to catch up with the primary instance's memory key-value pairs through replaying the Write-Ahead Log (WAL) in its memtable. On the other hand, Column Families allow the logical partitioning of KV-pairs within the same database instance. By leveraging these properties, SCID utilizes different logical partitions for KV-pairs with varying consistency levels. The database can be made multi-consistent on its secondary instances by modifying the call to catch up in Dynamic read-only instances to



(a) Current Systems



(b) Proposed System

Figure 4.2: Consistency Management in Current and Proposed System

`OnDemandSync(ColumnFamily)`, allowing one read-only instance to synchronize one column family while others remain unchanged. This design avoids the latency of synchronizing all data together.

This design allows for SCID to have an interesting property to allow data on every server to exhibit a different consistency level. While not implemented directly in SCID, this is notable as it allows for the same key-value pair to have different consistency guarantees on different servers based on the server's use case rather than the overall system requirement. However, our implementation does not dwell on this and is limited to a single consistency guarantee across all servers, leaving this as a future avenue for research.

4.2 Managing Performance-Consistency Trade-off via ML Model

Here we aim to discuss how we determine the consistency value of a particular key-value pair. The goal is to predict what consistency level would be appropriate for a particular KV-pair, this consistency level needs to take into account the accesses that are made to that particular data point along with factors like when and how the data is accessed. For such a solution where the parameters are not very well defined, and in some cases can be completely new to the machine. These scenarios will benefit from a Machine Learning model at the core that allows for predictions on data that is unseen in the past.

The ML model must meet two key requirements. Firstly, SCID accommodates client inputs in case of prediction errors, allowing clients to set a specific consistency level for any key-value pair on write or read, bypassing the ML model. Secondly, to minimize overhead and preserve database throughput, SCID employs models with lower run-time complexity, such as Naive Bayes, Random Forest, and Logistic Regression [29, 30, 24].

Due to the lack of columns in the key-value store, SCID tackles this challenge by utilizing metadata associated with the system and the key-value pairs to create additional columns. Eight features are gathered for each row of training data: Time, Date, Key, Value, Key Size, Value Size, Operation type, and Access Count. Since some of these features are non-numerical, Naive Bayes or Logistic Regression may suffer in accuracy. Random Forest performs better in this regard, demonstrating higher accuracy and lower latency while satisfying the second requirement.

SCID adopts the Random Forest model, a supervised learning technique. This requires an initial database with correct consistency-level assignments for training the model. Assigning values for each case can be done manually, varying according to the specific use case. Once the database is created and labeled, the Random Forest model can be integrated into the docker container on a separate thread, as depicted in Fig. 4.1.

SCID's integration of a machine learning model, particularly Random Forest, for consistency-level prediction introduces a powerful adaptive capability. This ensures that SCID can adapt to changing workloads and evolving data access patterns over time. As the system learns from historical data and client inputs, it becomes more adept at making precise consistency-level decisions, further bolstering overall system performance.

Chapter 5

IMPLEMENTATION

We developed the prototype of SCID based on RocksDB and use a combination of Kubernetes and Docker to achieve the dynamic management of multiple read-only instances. The use of Kubernetes as an orchestrator allows for partition tolerance and scalability, and Docker containers inside the cluster that run RocksDB internally to perform the Database operations. We use RocksDB (v7.4.5) and modified it to allow for connections using TCP/IP and created a new OnDemandSync() function call to partially sync selective consistency levels. Finally, Hadoop Distributed File System (HDFS) has been used at the persistent storage layer, HDFS allows for a singular storage location for data from all instances along with read streaming, distributed storage, and replication.

To obtain the baseline, the system uses the same method that existing work use (i.e., primary-secondary mode with data synchronization during the writes). The primary instance will wait for the secondary instances to replicate the data and once the requested number of instances. Once the requested number of instances has replicated the data, an acknowledgment is sent back to the user. When SCID is evaluated, the proposed system in the previous chapter is employed. The difference is when the data is made consistent when compared to the baseline. Compared to the baseline, SCID performs better in Writes and suffers in Reads which will be shown in Chapter 6.

The second implementation of SCID is related to the ML model. We address this implementation as SCID-ML and train a model using the Random Forest method as described in Chapter 4. We utilize a 100% read workload from YCSB as a testing

sample, using an input of 100,000 rows of data, using a 70:30, train:test split (Note: A newly generated workload is used for the actual benchmarking). The workload generated is generated in a Zipfian fashion and demonstrates the property of higher access counts for a small subset of data. We utilize this and remove the access count column, converting it into a label column with labels ranging from 1 to 5, equally distributed based on the access count. This dataset is now utilized to create a Random Forest model that is integrated into the setup. Note that since access count is how we define the consistency level, we do not provide that column to the ML model during any evaluations.

Due to the lack of available models that serve the purpose of predicting consistency, we use a random-prediction model as the baseline. As the name suggests the random-prediction model guesses a consistency value randomly, and the SCID-ML model uses the ML model created for SCID. To train this model, we use AutoGluon [25] and achieve an accuracy of 62%. In both situations, if the model predicts the incorrect value, we invoke the scenario where the client informs the system of the correct value, and the delay caused by this is taken into consideration when measuring throughput.

Chapter 6

EVALUATION

Based on the system design, SCID should improve on the write performance while having read performance that is worse in the worst case scenario. In this section, the same is tested and evaluated with various test cases. To assess the performance and effectiveness of the proposed system, we conducted evaluations using multiple read-write workloads and also multiple consistency levels of KV-pair read queries. The evaluation aims to measure the system's capabilities and address the following research questions:

- **Research Question 1:** How does the proposed consistency model (SCID) perform compared to the Baseline for various distributions of strong and eventual consistency in databases?
- **Research Question 2:** Does the ML model in SCID provide a performance uplift?

6.1 Test Setup and Metrics

All implementations have been performed on a Machine with 6 cores of an Intel(R) Xeon(R) Gold 6330 CPU, and 8 GB of RAM. A total of 3 instances are used to perform all tests, 1 primary and 2 secondary. This has been done to demonstrate the worst case scenarios for both models.

We further use three YCSB workloads with different write and read ratios for the tests; W (Write-only), A (50% writes, 50% reads), and C (Read-only), testing how the consistency models behave in different scenarios. All workloads have been run in

the Zipfian distribution consisting of 100,000 key-value pairs and 100,000 operations. We focus on the following measurement metrics: 1) throughput (query per second, the higher the better), 2) latency (milli-second, the lower the better).

6.2 SCID Performance

We first evaluate the consistency model against the baseline. Tests with varying distributions of strongly and eventually consistent data are performed. For every workload of YCSB mentioned, 100,000 keys were used with 0 to 100% ratio of strong and eventual consistency, with 10% intervals. These tests aim to comprehensively demonstrate how SCID will fare against the baseline in multiple scenarios. It should also be noted, that all conducted tests will consider the worst-case scenarios for read and writes. A read request comes directly after the write request, not allowing either database to do any background tasks after the writes are completed. A write request will also come right after the previous write request, not allowing for any background tasks after the first write request is completed.

6.2.1 Workload-based Tests

YCSB W - This test constitutes a 100% Write workload. Given the design of the consistency models, we expect SCID to have the same throughput and latency for any consistency distribution. SCID implements a methodology where Writes are acknowledged immediately and written to persistent media, and based on server workloads later propagated to other secondary instances. This allows SCID to guarantee data being persisted immediately in contrast to the Baseline model. On performing the tests, we see the expected results reflected in the resulting output. This is seen in Fig. 6.1 where SCID has lower latency and higher throughput than the Baseline as data gets more strongly consistent. Depending on the distribution of strong and eventually

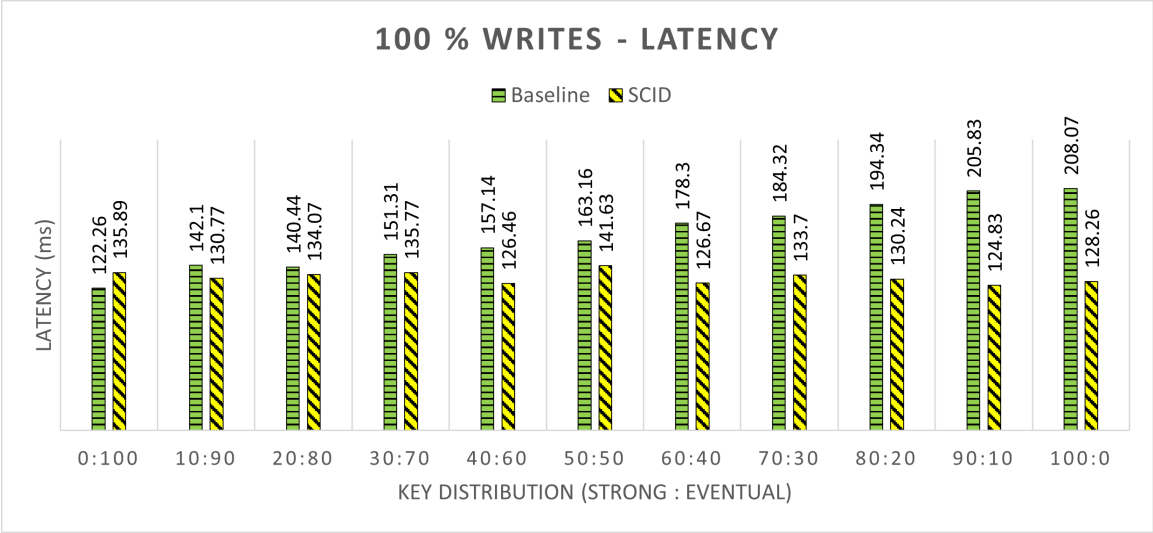
consistent data, we see SCID having up to 50% increase in throughput (Fig. 6.1b) compared to the Baseline.

The YCSB W test demonstrated the effectiveness of SCID in handling a 100% Write workload, with SCID showing up to a 50% increase in throughput compared to the Baseline model. This indicates that is a more efficient approach for handling writes across consistency levels.

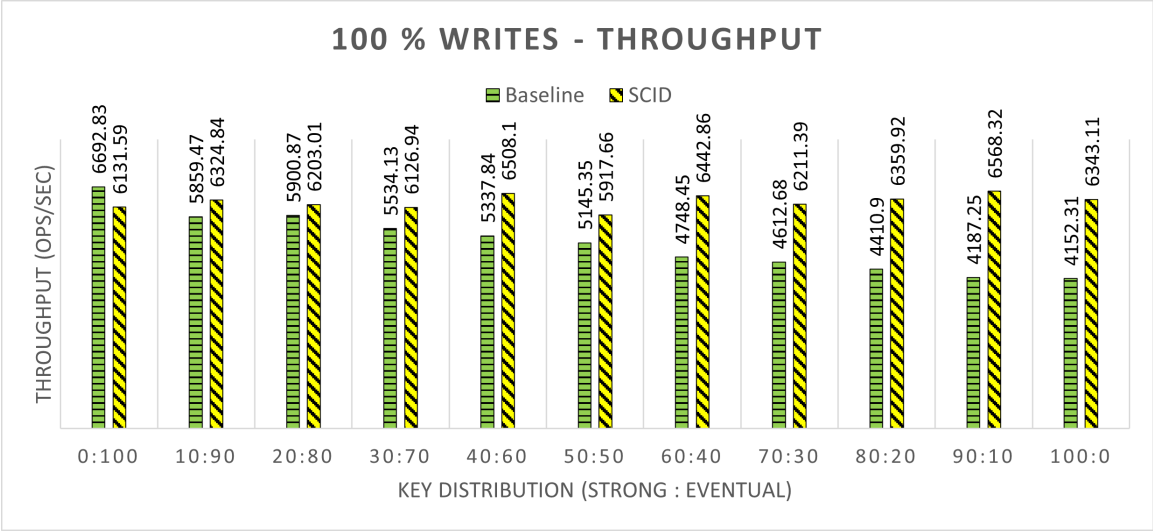
YCSB A - This test constitutes a 50% Write and 50% Read workload. Since both models have trade-offs, SCID in reads and Baseline in writes, we expect the results to be similar. This expectation is also met in the tests (Fig. 6.2) where both models have very similar latency (Fig. 6.2a) and throughput (Fig. 6.2b).

The YCSB A test has a mixed workload of Reads and Writes; it assumes the worst-case scenario for Reads and Writes both. The results extracted here point to SCID being at par with the current solutions, not losing performance even with different operations.

YCSB C - This test constitutes a 100% Read workload. As data gets more strongly consistent, we expect the Baseline model to outperform SCID. Here, we utilize another mechanism of SCID which allows the primary instance to respond to the values, however, this will come with a bandwidth limitation since only the primary instance will have the correct values. We call this implementation of SCID, SCID-Optimal, allowing for higher throughput but served from the primary instance only. When the secondary instance is forced to serve the reads, the implementation is called SCID-Offloaded. Here, the expectation of the baseline model outperforming SCID is expected because the baseline model will already have replicated the data in the loading stage. Since the tests consider the worst-case scenarios only (reads happen right after the write), SCID-Offloaded will not have the data present at the secondary instances. This difference is reflected in the results in Fig. 6.3 and leads to

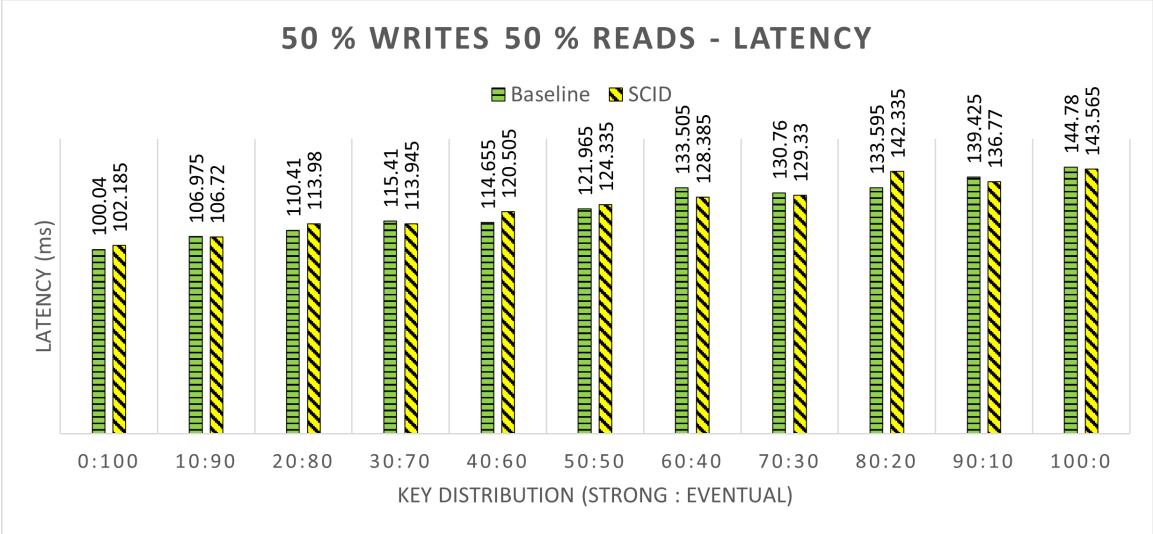


(a) Latency

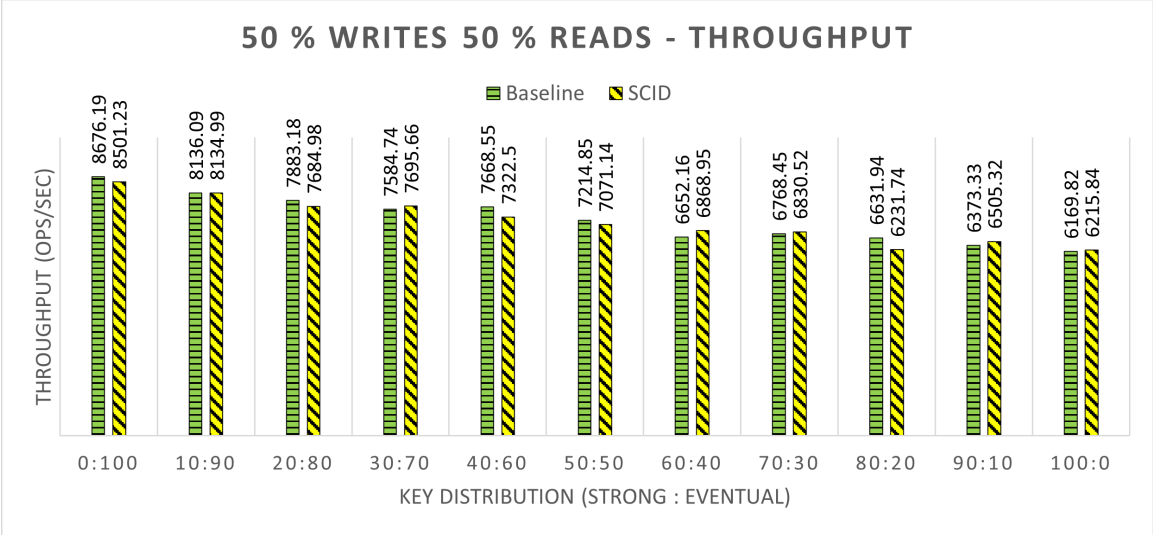


(b) Throughput

Figure 6.1: YCSB W - 100% Writes



(a) Latency



(b) Throughput

Figure 6.2: YCSB A - 50% Writes, 50% Reads

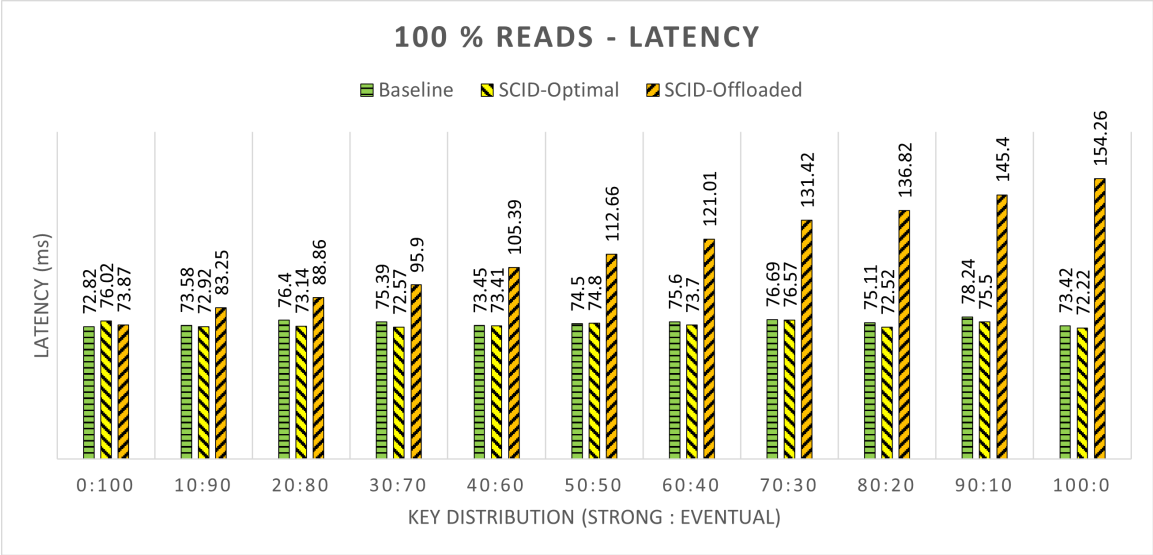
SCID-Offloaded having higher latency (Fig. 6.3a) and lower throughput (Fig. 6.3b) by at much as 50%; and SCID-Optimal having almost equivalent throughput and latency to the Baseline.

The YCSB C test considers the situation for 100% read operations. This test contains SCID-Optimal and SCID-Offloaded, where both situations, when SCID primary and secondary instances serve reads respectively. The results here point to SCID-Optimal being at par with the current solutions and SCID-Offloaded having at least 50% throughput of current solutions.

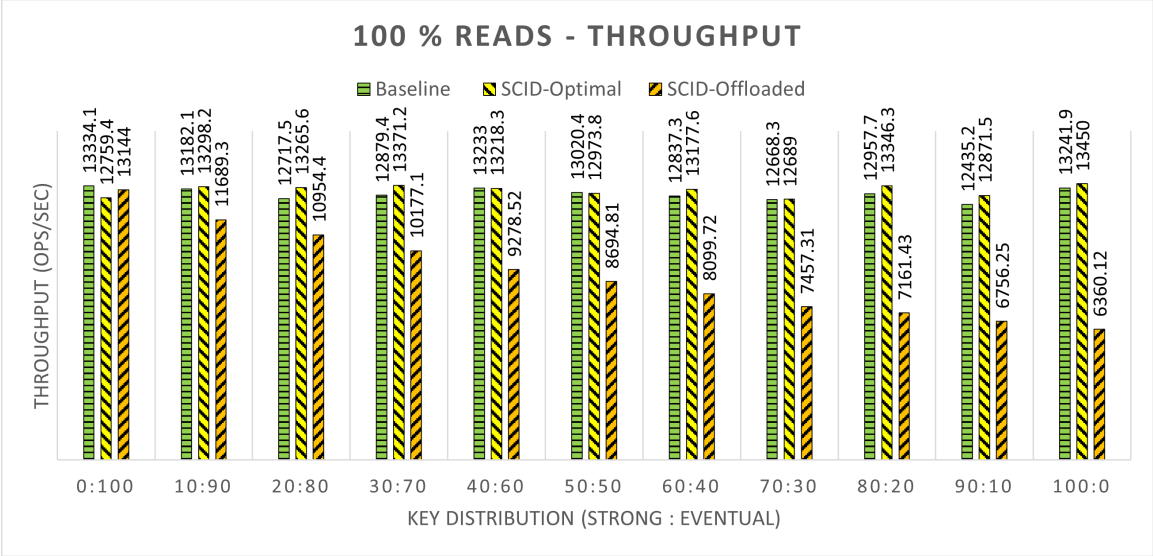
6.2.2 Consistency-based Tests

The Workload-based tests demonstrate that as data gets more strongly consistent, we can see a more distinctive image of how SCID differs from the baseline. We further create more workloads, focusing specifically on only 100% Strong consistency and evaluate the difference between SCID and the baseline model. The result for this evaluation can be seen in Fig. 6.4, while the Baseline has a throughput that suffers in any write-heavy scenario, it gets much better in a read-heavy scenario. SCID in the meantime has consistent performance across the board, showing similar throughput (Fig. 6.4b) and latency (Fig. 6.4a) regardless of the workload. This would be beneficial in case of any scenario where the data is not read immediately after write, allowing SCID to have near-equal throughput in reads and making the offering much better.

The Consistency based tests take the worst-case scenarios at 100% strong consistency and show that SCID maintains the throughput regardless of the workload. This is in contrast to the baseline model, which can have a higher or lower throughput based on the workload that is being utilized.

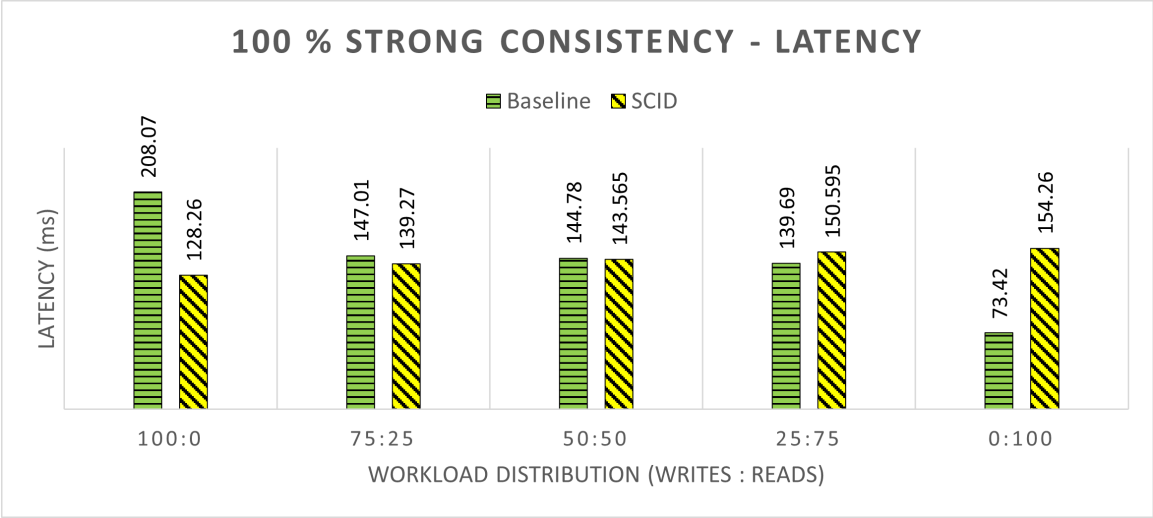


(a) Latency

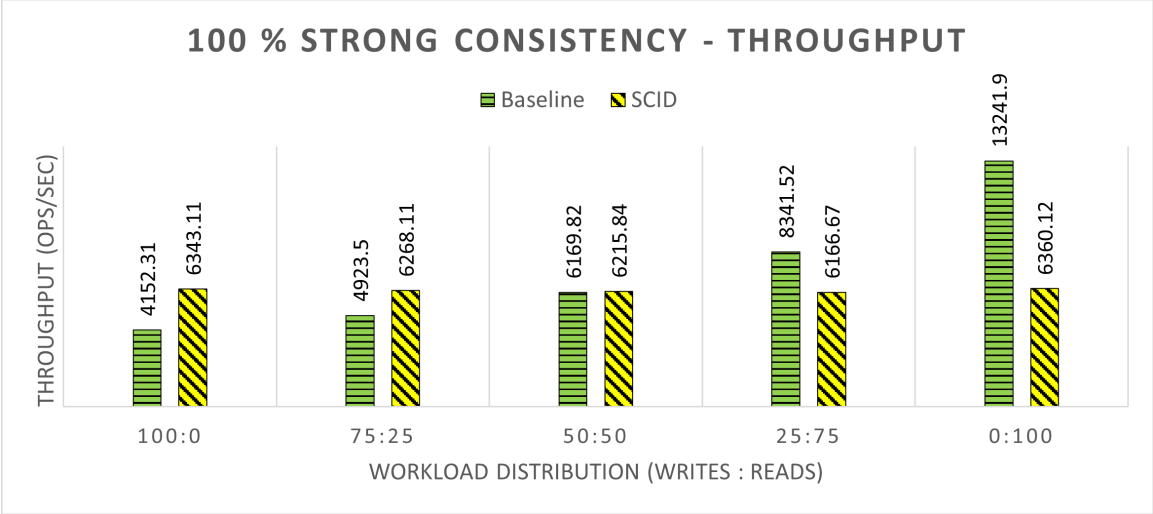


(b) Throughput

Figure 6.3: YCSB C - 100% Reads



(a) Latency



(b) Throughput

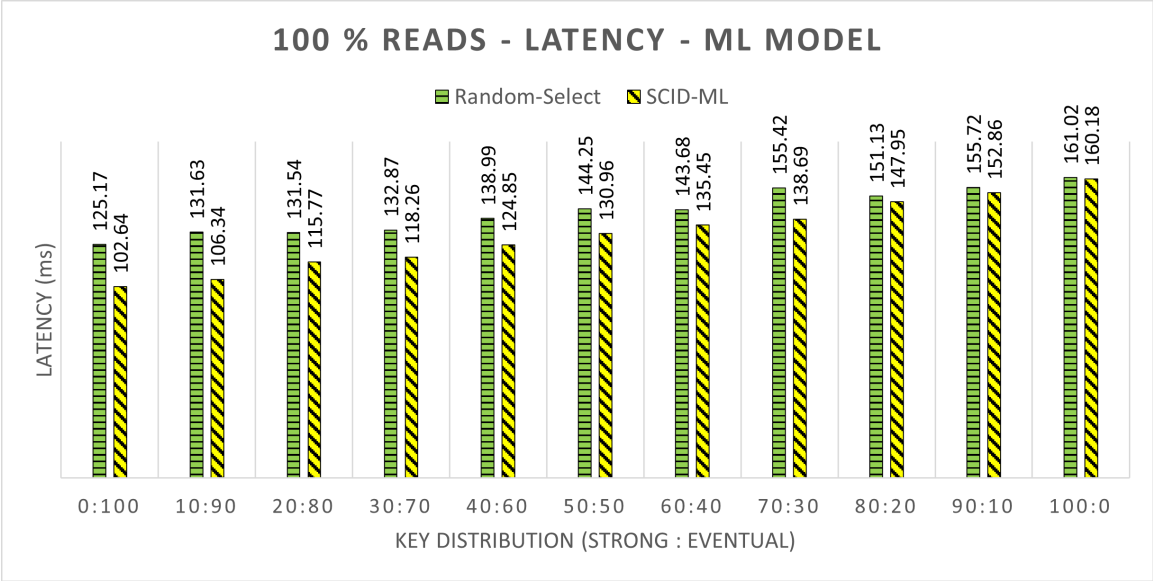
Figure 6.4: 100% Strong Consistency - Various Workloads

6.3 SCID-ML Performance

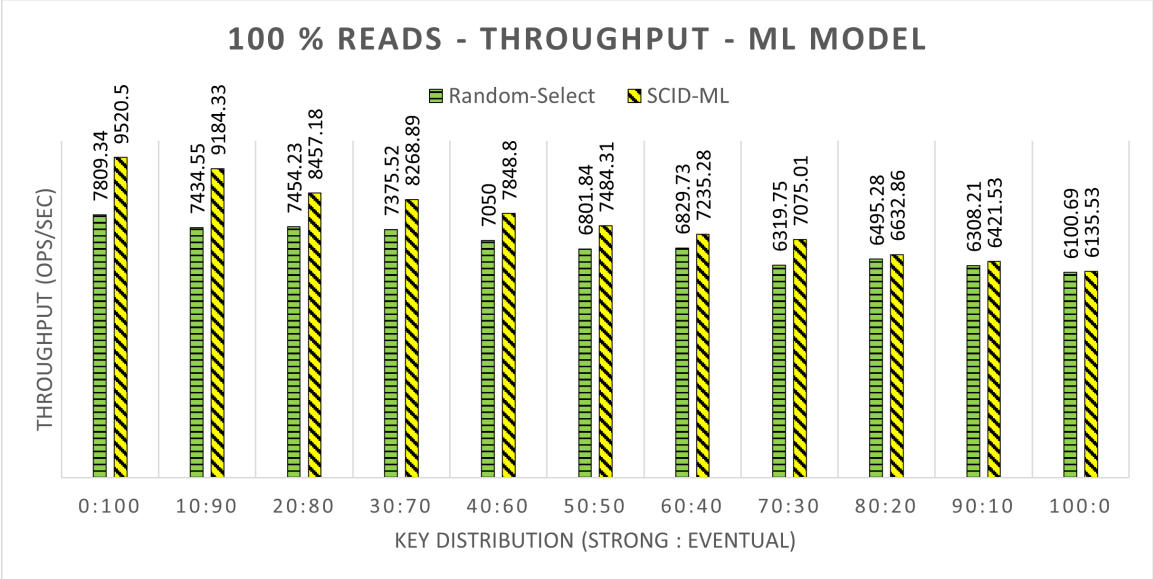
Another set of tests with various distributions of consistency is carried out to measure the performance of the ML model. To sufficiently compare the model and cover the lack of pre-existing models that predict consistency, we use a random-consistency baseline. The random consistency model randomly guesses the consistency for a particular key-value pair. To evaluate SCID, the ML model trained for SCID is used to generate the consistency level. When either model provides a value for consistency, the code performs the corresponding action for the consistency level, and if it is then found that the wrong consistency is assigned, the case where the client provides the correct consistency is invoked and the appropriate actions are taken as described in the design section.

Using this information, it can be expected that for any ML model that performs well and correctly, it can outperform a model that randomly guesses data. This is demonstrated in Fig. 6.5 where it can be observed that as data is more often strongly consistent, both models have about equal latency and throughput, but as the distribution of consistency starts to move towards eventual, the ML model outperforms the random-consistency model. This lines up with our expectation as the ML model has an accuracy of 62% which makes it more likely to correctly guess the consistency than the random-consistency model.

Taking the ML tests as a separate unit, it is observed that SCID will outperform the random-selection model by 21%, with an accuracy of 62%. Utilizing the model can assist the developers to get better estimates and speed up the development process.



(a) Latency



(b) Throughput

Figure 6.5: 100% Read Workload - ML Model Evaluation

Chapter 7

DISCUSSION AND CONCLUSION

This thesis introduces a new methodology to maintain consistency in databases. Performing replications in the background and before reads rather than instantly on writes. It utilizes various aspects of LSM-KVS to perform this and presents the solution SCID. With SCID, we improve in scenarios where the databases are write and update heavy. Overall, SCID is an approach to consistency that trades immediate read performance for improved write performance. We further present an ML-based approach to assigning consistency values in Key-Value stores that relies upon a combination of metadata and the key values to predict the consistency level of given data.

The approaches are evaluated and show the major advantages and disadvantages of SCID. Taking the worst-case situations into consideration, in a 100% Write workload, SCID maintains its throughput and latency. In a 100% Read situation, the model when performing reads from the secondary instance suffers in comparison to the current implementation, however, if the reads are performed from the primary instance directly, the performance penalties are reduced. When evaluating the ML model, there is a lack of other baselines to compare against, we devise our baseline, one that utilizes random-value assignment to consistency levels to have a better estimation of consistency levels. The model also has another solution that allows for forcing a particular consistency level directly to the database which can differ from what the model predicts. This model can prove to be a great aid for developers when the same database is used by multiple teams, aiding the process of consistency-level assignment to key-value pairs.

Considering the consistency model, SCID has trade-offs when compared with the baseline model. However, these trade-offs disappear once we take SCID-Optimal into consideration which serves reads directly from the primary at the cost of lower bandwidth of the primary instance, which is not desirable for instances when mixed workloads are present. A future path would be the efficient use of SCID-Optimal, reducing the bandwidth by integrating SCID-Offloaded for parts of the database. We can also further improve the ML model by performing optimizations on the actual model and exploring more optimized models that can more effectively predict consistency levels.

Another avenue of research can be to have more effective benchmarking systems where multiple consistency levels are taken into account. While YCSB provides for various workloads, it does not allow for testing with multiple consistency levels at the same time. A more robust framework for benchmarking such systems can be developed and used to effectively test such systems.

Overall, we note that SCID allows further optimizing the consistency-performance trade-off, and presents a methodology that allows circumventing this trade-off providing a flexible middle ground. While SCID may not apply to every situation, especially when quick responses are needed immediately after writes, it offers much for situations where there is a majority of eventual consistency data or where there is a lack of data being immediately accessed after writes.

REFERENCES

[1] "Revisiting B+-tree vs. LSM-tree", URL https://www.usenix.org/publications/loginonline/revisit-b-tree-vs-lsm-tree-upon-arrival-modern-storage-hardware-built (2022).

[2] "B-Tree vs Log-Structured Merge-Tree - Deep Dive TiKV", URL https://tikv.github.io/deep-dive-tikv/key-value-engine/B-Tree-vs-Log-Structured-Merge-Tree.html (2023).

[3] "BadgerDB: Fast key-value DB in Go.", URL https://github.com/dgraph-io/badger (2023).

[4] "Column Families", URL https://github.com/facebook/rocksdb/wiki/Column-Families (2023).

[5] "Data protection in Amazon S3 - Amazon Simple Storage Service", URL https://docs.aws.amazon.com/AmazonS3/latest/userguide/DataDurability.html (2023).

[6] "Dynamo | Apache Cassandra Documentation", URL https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html#tunable-consistency (2023).

[7] "etcd", URL https://etcd.io/ (2023).

[8] "FollowFeed: LinkedIn's Feed Made Faster and Smarter", URL https://engineering.linkedin.com/blog/2016/03/followfeed--linkedin-s-feed-made-faster-and-smarter (2023).

[9] "HDFS Architecture Guide", URL https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (2023).

[10] "Production-Grade Container Orchestration", URL https://kubernetes.io/ (2023).

[11] "Read only and Secondary instances", URL https://github.com/facebook/rocksdb/wiki/Read-only-and-Secondary-instances (2023).

[12] "Tunable Consistency and Consistency Levels in Apache Cassandra®", URL https://www.datastax.com/learn/cassandra-fundamentals/tunable-consistency (2023).

[13] "Write Concern — MongoDB Manual", URL https://www.mongodb.com/docs/manual/reference/write-concern/ (2023).

[14] "RedisTair_ Redis -", URL https://help.aliyun.com/document_detail/145957.html (2023).

[15] alastairn, "Cassandra Column Family", URL https://www.scylladb.com/glossary/cassandra-column-family/ (2023).

[16] Atikoglu, B., Y. Xu, E. Frachtenberg, S. Jiang and M. Paleczny, "Workload analysis of a large-scale key-value store", in "Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems", SIGMETRICS '12, pp. 53–64 (Association for Computing Machinery, New York, NY, USA, 2012), URL https://doi.org/10.1145/2254756.2254766.

[17] Balegas, V., S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh and M. Shapiro, "Putting consistency back into eventual consistency", in "Proceedings of the Tenth European Conference on Computer Systems", pp. 1–16 (ACM, Bordeaux France, 2015), URL https://dl.acm.org/doi/10.1145/2741948.2741972.

[18] Blog, N. T., "Application data caching using SSDs", URL https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef (2017).

[19] Burckhardt, S., "Consistency in Distributed Systems", in "Software Engineering: International Summer Schools, LASER 2013-2014, Elba, Italy, Revised Tutorial Lectures", edited by B. Meyer and M. Nordio, Lecture Notes in Computer Science, pp. 84–120 (Springer International Publishing, Cham, 2015), URL https://doi.org/10.1007/978-3-319-28406-4_4.

[20] Cao, Z., S. Dong, S. Vemuri and D. H. C. Du, "Characterizing, Modeling, and Benchmarking {RocksDB} {Key-Value} Workloads at Facebook", pp. 209–223 (2020), URL https://www.usenix.org/conference/fast20/presentation/cao-zhichao.

[21] Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data", in "7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)", pp. 205–218 (2006).

[22] Choudhury, S., "Apache Cassandra: The Truth Behind Tunable Consistency, Lightweight Transactions & Secondary Indexes", URL https://www.yugabyte.com/blog/apache-cassandra-lightweight-transactions-secondary-indexes-tunable-consistency/ (2018).

[23] Cooper, B. F., A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears, "Benchmarking cloud serving systems with YCSB", in "Proceedings of the 1st ACM symposium on Cloud computing", SoCC '10, pp. 143–154 (Association for Computing Machinery, New York, NY, USA, 2010), URL https://doi.org/10.1145/1807128.1807152.

[24] Crankshaw, D., "The Design and Implementation of Low-Latency Prediction Serving Systems | EECS at UC Berkeley", URL https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-171.html (2023).

[25] Erickson, N., J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li and A. Smola, "AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data", URL http://arxiv.org/abs/2003.06505, arXiv:2003.06505 [cs, stat] (2020).

[26] Facebook Database Engineering Team, "RocksDB: A Persistent Key-Value Store for Flash and RAM Storage", URL https://github.com/facebook/rocksdb/ (2023).

[27] Ghemawat, S. and J. Dean, "LevelDB", URL https://github.com/google/leveldb (2023).

[28] Gilbert, S. and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", ACM SIGACT News **33**, 2, 51–59, URL https://dl.acm.org/doi/10.1145/564585.564601 (2002).

[29] Hirschman, J., A. Kamalov, R. Obaid, F. H. O'Shea and R. N. Coffee, "At-the-Edge Data Processing for Low Latency High Throughput Machine Learning Algorithms", in "Accelerating Science and Engineering Discoveries Through Integrated Research Infrastructure for Experiment, Big Data, Modeling and Simulation", edited by K. Doug, G. Al, S. Pophale, H. Liu and S. Parete-Koon, Communications in Computer and Information Science, pp. 101–119 (Springer Nature Switzerland, Cham, 2022).

[30] Kumar, P., "Time Complexity of ML Models", URL https://medium.com/analytics-vidhya/time-complexity-of-ml-models-4ec39fad2770 (2021).

[31] Lamport, L., "Paxos Made Simple", ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) pp. 51–58, URL https://www.microsoft.com/en-us/research/publication/paxos-made-simple/ (2001).

[32] Li, C., J. Leitão, A. Clement, N. Preguiça, R. Rodrigues and V. Vafeiadis, "Automating the Choice of Consistency Levels in Replicated Systems", pp. 281–292 (2014), URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2.

[33] Li, C., D. Porto, A. Clement, J. Gehrke, N. Preguiça and R. Rodrigues, "Making {Geo-Replicated} Systems Fast as Possible, Consistent when Necessary", pp. 265–278 (2012), URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

[34] Maddox, T., "Research: 68% report cost is biggest data storage pain point", URL https://www.techrepublic.com/article/research-68-report-cost-is-biggest-data-storage-pain-point/ (2015).

[35] McConnon, A., "How we built a general purpose key value store for Facebook with ZippyDB", URL https://engineering.fb.com/2021/08/06/core-data/zippydb/ (2021).

[36] Nishtala, R., H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung and V. Venkataramani, "Scaling Memcache at Facebook", pp. 385–398 (2013), URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[37] Ongaro, D. and J. Ousterhout, "In Search of an Understandable Consensus Algorithm", pp. 305–319 (2014), URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[38] Pang, Z., Q. Lu, S. Chen, R. Wang, Y. Xu and J. Wu, "ArkDB: A Key-Value Engine for Scalable Cloud Storage Services", in "Proceedings of the 2021 International Conference on Management of Data", SIGMOD '21, pp. 2570–2583 (Association for Computing Machinery, New York, NY, USA, 2021), URL https://doi.org/10.1145/3448016.3457553.

[39] Phansalkar, S. P. and A. R. Dani, "Tunable consistency guarantees of selective data consistency model", Journal of Cloud Computing **4**, 1, 13, URL https://doi.org/10.1186/s13677-015-0038-4 (2015).

[40] Qiao, Y., X. Chen, N. Zheng, J. Li, Y. Liu and T. Zhang, "Closing the B+-tree vs. {LSM-tree} Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression", pp. 69–82 (2022), URL https://www.usenix.org/conference/fast22/presentation/qiao.

[41] Schultz, W., T. Avitabile and A. Cabral, "Tunable consistency in MongoDB", Proceedings of the VLDB Endowment **12**, 12, 2071–2081, URL https://doi.org/10.14778/3352063.3352125 (2019).

[42] seesharprun, "Consistency level choices - Azure Cosmos DB", URL https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels (2023).

[43] Shapiro, M., N. Preguiça, C. Baquero and M. Zawirski, "Conflict-free Replicated Data Types", vol. 6976, p. 386 (Springer, 2011), URL https://hal.inria.fr/hal-00932836.

[44] Sivaramakrishnan, K., G. Kaki and S. Jagannathan, "Declarative programming over eventually consistent data stores", ACM SIGPLAN Notices **50**, 6, 413–424, URL https://doi.org/10.1145/2813885.2737981 (2015).

[45] Vora, M. N., "Hadoop-HBase for large-scale data", in "Proceedings of 2011 International Conference on Computer Science and Network Technology", vol. 1, pp. 601–605 (2011).

[46] Wang, J., C. Li, K. Ma, J. Huo, F. Yan, X. Feng and Y. Xu, "AUTOGR: automated geo-replication with fast system performance and preserved application semantics", Proceedings of the VLDB Endowment **14**, 9, 1517–1530, URL https://doi.org/10.14778/3461535.3461541 (2021).