A Dependency Tracking Storage System for Optimistic Execution of Serverless Applications

by

Suraj Singh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

Statement of Contributions

I would like to acknowledge Senyu Fu for contributing to the research described in this thesis. Senyu helped implement components of the Arbor system. Any components for which Senyu was primarily responsible for have been omitted from this thesis.

Abstract

Serverless computing has become an increasingly popular paradigm for building cloud applications. There has been a recent trend of building stateful applications on top of serverless platforms in the form of workflows composed of individual functions. As functions are short-lived and state is not recoverable across function invocations, these applications typically store state that is used between functions in an external storage system. Such storage systems should enforce concurrency control, as different workflow instances may update overlapping state simultaneously. However, existing concurrency control algorithms typically incur significant latency due to locking or read/write set validation. This is undesirable, since execution latency is an important performance metric for workflow applications as each stage is executed sequentially. Furthermore, they can abort transactions in a manner that is oblivious to application preferences.

In this thesis, we present Arbor, a sharded dependency-tracking storage system designed for optimistic execution of serverless workflows while ensuring serializability. Arbor introduces a two-round commit model where submitted client transactions are organized in a dependency graph. Transactions are then processed in batches, off the critical path of client execution, allowing clients to continue executing quickly without having to wait for Arbor to validate each transaction. As Arbor processes transactions, it organizes them into a tree where each branch is a serialized execution and conflicts result in new branches being created. It then commits one branch from this tree and prunes the rest. To minimize re-executions, Arbor chooses the longest branch by default, but application developers can implement their own policies. Pruning branches is simple with Arbor, since it can re-execute the corresponding transactions by invoking the respective functions from the serverless platform. Furthermore, Arbor is designed to be scalable. Data is partitioned by key, but the metadata of its dependency graph is replicated. This design allows single-shard transactions in each batch to be processed independently, while multi-shard transactions are replicated and processed by each shard. Our evaluation on a cluster of machines shows that Arbor's two-round commit model reduces transaction execution latency by a median value of 1.26x when compared to a system that uses OCC and commits transactions synchronously.

Acknowledgements

I thank my supervisors Professor Bernard Wong and Professor Khuzaima Daudjee for their guidance and support during my graduate studies. Khuzaima has always encouraged me to think about the bigger picture and question the ‘why’ behind every design decision. This has helped me arrive at a more focused and well-thought-out approach to research. Bernard’s out-of-the-box approach to solving problems has inspired me to think of creative solutions to the technical problems we faced along the way. Furthermore, both Khuzaima’s and Bernard’s detailed feedback and comments have helped improve my technical writing significantly. Through describing many of their own experiences and using some remarkable analogies, they have helped me cultivate a resilient mindset to tackle problems in life. I am extremely grateful for the experience of working with them.

I thank the readers of my thesis, Professor Ken Salem and Professor Ali Mashtizadeh for their valuable comments on this work. I am also thankful to Senyu Fu for his collaboration on this project. I have learned a lot from our discussions in the lab and reasoning about the various bugs we encountered along the way. These experiences helped in bringing this thesis to completion. I would also like to thank Dr. Xinan Yan. He was always willing to listen to my ideas and gauge whether they made sense. He helped significantly in giving this project a clear direction. I’d also like to thank other members of my lab and the Shoshin group with whom I’ve had a great experience working with and that helped create a positive environment.

With my program beginning in the middle of a global pandemic, my friends played a pivotal role in my life as a graduate student. In particular, I’d like to thank Lasantha, James, Kerem, Saksham, Sruthi, Matt, Joohan, Saraswathi, Mahir, and Sachit. I will cherish our experiences and am grateful for them all. I also thank the staff at CSCF and the CS Graduate Office for their help along the way. Carrying out this work would not have been possible without them.

Finally, I want to thank my family for their constant support and love throughout my graduate studies. They have always been there to listen, give advice and just talk to me when I’ve needed it. I would especially like to thank my mother Archana for her love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Serverless computing has become an increasingly popular paradigm for building cloud applications. In traditional 'severful' cloud platforms, developers must manually provision resources (for example, AWS EC2 virtual machines). This can be problematic, as it can lead to over-provisioning resources to account for bursty user traffic or under-provisioning them and causing application slowdown [27]. On the other hand, serverless platforms provide developers with a much simpler abstraction in which they can upload application code as functions that end-users can invoke through various events (for example, uploading an image to AWS S3 [1] or accessing a URL). Resources are provisioned and automatically scaled by cloud vendors according to invocation load. Moreover, serverless platforms offer a millisecond-granular pay-per-use billing model instead of having developers pay for the resources provisioned. These aspects allow developers to deploy applications more easily and at a lower cost.

Applications developed as stateless functions are particularly well suited to serverless platforms. For example, a serverless function can be used in a social media application to identify objects within an image and create thumbnails for different platforms (such as desktop and mobile) [27, 11, 3]. Since such functions can be invoked independently, cloud vendors can easily scale compute resources according to the invocation load, making them an excellent choice for serverless platforms. Developing more complex applications that involve manipulating global state or those that involve the coordination of multiple functions is challenging because: (1) state is not persistent across multiple invocations of the same function, (2) functions are not network addressable, preventing direct inter-function communication and (3) functions have strict execution time limits [24]. However, there is still an incentive to enable the execution of such complex applications on serverless platforms due to the benefits of autoscaling resources and lower operating costs.

As a workaround, developers use external storage systems (such as AWS S3 and DynamoDB [17]) to store application state and also as an intermediary to propagate output between functions coordinating with each other [27]. Cloud providers formalized this approach through the introduction of serverless orchestration frameworks (such as AWS Step Functions [4]) [24]. Such frameworks allow developers to compose workflows of functions through the declaration of task graphs. Developers can embed conditional logic and create cycles within a graph to represent loops. As an example of a stateful serverless workflow, consider a social media website where a user can log in, view their timeline, and follow/unfollow other users — each action can be implemented as a separate function in a workflow that accesses/updates state stored in an external storage system [48, 22]. When such a workflow is invoked, the cloud provider takes care of provisioning the required resources and transferring the intermediate state between functions.

However, using external storage systems to enable the execution of stateful applications on serverless platforms incurs significant latency. According to Hellerstein et al., [24] the mean I/O latency (sequential write followed by read) for 1 KB of data for AWS S3 and DynamoDB from an AWS Lambda function is 108 ms and 11 ms, respectively. Transactional storage systems are particularly useful to support serverless workflows since functions can make updates to global application state and consequently result in incorrect computation due to race conditions [36]. Due to their strong consistency guarantees, transactional storage systems prevent these issues and make it easier to develop stateful applications. Transactional storage systems can, however, exacerbate latency overhead for stateful applications since concurrency control often involves acquiring locks or analyzing read/write sets of transactions. Such systems also commonly use distributed commit protocols such as two-phase commit (2PC), which add further overheads due to inter-shard coordination [15]. Application throughput is adversely affected by such overheads since interacting with storage systems is on the critical path of execution.

Several systems have been designed for stateful serverless applications [39, 25, 50, 48]. Boki [25] is a serverless platform developed on top of a shared log abstraction [9]. It introduces the novel concept of a 'metalog' which allows it to provide strong consistency, fault tolerance, and high scalability. However, transaction execution takes several milliseconds when using Boki's shared log as a durable object store and can consequently limit application performance. Moreover, Boki's shared log increases execution latency by up to 3x when providing exactly-once semantics and transactional guarantees for serverless workflows using DynamoDB to store state.

In OLTP applications that can have a high degree of contention, validating transactions on the critical path of client execution is preferable to limit the number of cascading aborts. However, serverless applications generally consist of multiple workflows, each requiring varying levels of consistency. For example, a social media application may include workflows for updating user profile information (independent state updates), recommending other users to follow (analytics pipelines), and allowing users to create posts or unfollow/follow others (shared state updates). Such 'hybrid' workloads thus require a storage system that provides serializability while not adding significant overhead to applications since a lot of the operations are non-conflicting.

This thesis presents Arbor, a sharded dependency-tracking storage system designed for optimistic execution of serverless workflows while ensuring serializability. To this end, Arbor introduces a novel two-round commit model that splits the transaction commit phase into *graph-commit* and *finalization*. As clients execute transactions, they can embed dependency information into updates representing the data used to compute the corresponding values. Arbor uses this information to organize data from client-submitted transactions in a dependency graph. This graph-commits each transaction and allows functions within a workflow to continue their execution quickly without waiting for Arbor to validate if transactions are serializable.

Arbor validates transactions in batches as part of a finalization process. Here, Arbor reorganizes transactions from its dependency graph into a validation tree, where each branch represents a serialized execution of transactions. If Arbor detects conflicts between transactions, it creates multiple branches of execution [16]; subsequent transactions are placed on the same branch as their ancestors. After adding all transactions in batch to the tree validation tree, as in blockchain-based systems [35], Arbor selects one branch of transactions to commit and prunes the rest. By default, Arbor chooses the longest branch to minimize re-executions, but it provides application developers with an interface to implement custom branch selection policies. It re-executes transactions along pruned branches by invoking the corresponding workflows from the serverless platform. Re-executed transactions may be pruned again subsequently if they are not on the branch that Arbor commits. In this case, Arbor will continue to re-execute them until they are.

Arbor's approach of taking transaction processing off the critical path of client execution can allow applications to execute quickly. This is particularly beneficial in the case of serverless applications since each stage in a workflow executes sequentially; faster execution of each stage allows for the subsequent stages to begin earlier and can consequently improve application throughput. The increased completion latency may make Arbor unsuitable for user-facing applications but this is a common characteristic of batch processing systems and can work well for applications where throughput is a priority and individual transactions are not latency sensitive. Furthermore, as in the case of the 'hybrid' workloads described earlier, contention in serverless applications can be limited. As a result, abort rates with Arbor would be low, limiting the overhead of re-executing transactions.

Arbor's dependency-tracking data model allows it to create branches in response to conflicts when processing transactions. This mechanism in turn gives applications more control over conflict resolution than standard approaches such as OCC. Committing the longest branch of transactions, for example, can reduce the number of re-executions. Alternatively, suppose in a social media application, there is a conflict over which posts to display on a user's timeline. In that case, the application developer can implement a policy to pick the branch whose engagement is estimated to be the highest. However, this dependecy-tracking data model introduces performance challenges as well. First, achieving scalability through partitioning is non-trivial since each partition would hold a different subset of the dependency graph. Second, unlike other transaction processing systems [33], committing transactions within Arbor's dependency graph is an intrinsically serial process and can consequently limit throughput since the system must process transactions in their topological order.

Arbor partitions data items by key, allowing it to scale with incoming graph-commit requests. However, this results in each shard holding a different subset of the dependency graph. When processing each batch of transactions, Arbor replicates the corresponding metadata of the dependency graph (i.e., keys and edges) across all shards in the deployment. Arbor's commit protocol allows transactions that include keys from only a single shard to commit independently on that shard. Cross-shard transactions, on the other hand, are replicated across all shards and committed by all of them. The commit order of single-shard transactions is disseminated across the shards and is deterministically aggregated by each of them before cross-shard transactions within a batch are processed. This allows each shard to arrive at the same state at the end of processing each batch of transactions. This protocol for committing transactions allows Arbor to scale well in throughput with the number of shards in workloads with a high percentage of single-shard transactions.

To optimize transaction processing, we implement a multi-stage pipeline where Arbor can process multiple batches of transactions concurrently. The pipeline includes stages for topologically sorting transactions, performing concurrency control, and aggregating results from all data shards, respectively. Furthermore, to make detecting conflicts more efficient, we implement a data indexing scheme called *lineage addressing* inspired by the fork paths used in TARDiS [16]. We assign each transaction in Arbor's validation tree a lineage address that uniquely identifies its position in the tree. Arbor leverages lineage addressing to efficiently detect conflicts between transactions and identify which branches to add future transactions to.

When functions within a workflow graph-commit transactions, their descendants operate on potentially dirty data since Arbor may abort the corresponding transactions during finalization. Thus, transactions reading graph-committed data operate at the *read un-*

committed isolation level [12] depending on the read policy that clients use. There is a possibility of cascading aborts, but if the level of contention is moderate, this is a reasonable trade-off since most transactions are likely to succeed. Using the YCSB+T workload [18], our experiments show that even with moderate levels of contention (Zipfian coefficient = 0.7), abort rates remained below 6%.

This thesis makes four main contributions:

- We introduce a novel two-round transaction commit model which enables faster execution of application clients by taking transaction processing off the critical path of client execution.
- We use a dependency-tracking data model to serialize transactions along different branches of a validation tree. We provide an interface for developers to implement custom policies for selecting which branch of transactions from the tree to commit.
- To achieve scalability in transaction processing, we introduce an architecture in which data is partitioned by key, and metadata of Arbor's dependency graph (i.e., keys and edges) is replicated across all shards. This allows each shard to commit single shard transactions independently, while cross-shard transactions are committed by all shards.
- We present an experimental evaluation on Arbor to understand the impact of different access patterns and levels of skew. Additionally, we investigate Arbor's scalability by gauging the effect of adding more shards and the impact of cross-partition transactions. Finally, we investigate the latency of graph-committing transactions in different types of workloads and compare it to a system that commits transactions synchronously.

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of the related work to this thesis. Chapters 3 and 4 describe the execution model and design of Arbor, respectively. Chapter 5 explains the experimental methodology and presents our evaluation of Arbor. Chapter 6 concludes the thesis and presents avenues for future work.

Chapter 2

Background and Related Work

This chapter will first provide a background on serverless computing and describe related work on storage systems used in the space. Next, it will describe related dependency-tracking storage systems. Finally, it will discuss the use of batching in database systems.

2.1 Serverless Computing

Cloud computing has greatly simplified the deployment of applications at scale. By virtualizing physical resources, cloud platforms enable developers to provision virtual machines with the resources they require and develop applications just as they would locally. However, user traffic can fluctuate [42], forcing developers to make trade-offs when provisioning resources. If they allocate resources to account for bursts in traffic, it would result in underutilization during other times and make them pay superfluous costs. Conversely, if they only provision resources based on their expected mean traffic, their costs would be relatively lower. Still, their infrastructure would be unable to handle sudden bursts of traffic, resulting in application crashes or high latency. Furthermore, developers are responsible for manually maintaining the resources they have reserved by updating their software and installing the latest security patches.

With the introduction of AWS Lambda [2] in 2015, serverless computing has solved many of these challenges. Serverless platforms provide developers with a much simpler abstraction where they can upload application-level code in the form of functions triggered based on events. Vendors automatically scale functions according to the invocation load. Furthermore, serverless platforms offer millisecond-granular pay-per-use billing. These aspects allow developers to deploy applications more easily and at a lower cost.

Serverless platforms are particularly well suited to execute stateless function invocations. An example of such an application would be a function for resizing images to thumbnails for different platforms. (for example, desktop and mobile) [3]. Such applications can take advantage of the autoscaling capabilities of serverless platforms. However, serverless platforms have the following disadvantages, which limit their applicability to other applications that require coordination between functions or state to be persisted: (1) state is not persistent across multiple invocations of the same function, (2) functions are not network addressable, preventing direct inter-function communication and (3) functions have strict execution time limits [24].

As a workaround, developers began using storage systems to store ephemeral state that is propagated between functions and for persisting state that is required across different function invocations [27]. Cloud vendors formalized this approach with the introduction of serverless orchestration frameworks (such as AWS Step Functions [4]. Users can compose workflows of functions (such as microservice-style applications [22, 48] by registering a task graph with the frameworks. Subsequently, the orchestration frameworks take care of provisioning resources and propagating state between functions. However, using external systems to facilitate such workflows can incur significant latency for the application. For example, Hellerstein et al. [24] state that the mean I/O latency (sequential write followed by read) from an AWS Lambda function to S3 and DynamoDB are 108 ms and 11 ms, respectively. As a result, optimizing for such overheads has been an important topic of research.

Pocket [30] was one of the first systems built for facilitating the propagation of ephemeral state between functions. It was designed specifically for use in big data analytics applications built on serverless platforms. Pocket's controller allows developers to register their jobs a priori, and it can significantly reduce running costs by allocating resources according to the job's requirements. Jiffy [29] is an improvement over Pocket and can reduce job completion time by multiplexing multiple jobs on main memory to reduce the number of accesses to slower persistent storage. Furthermore, it can dynamically change the job's storage allocation during runtime based on active data being used and as a result, further save running costs. Ephemeral storage systems such as Pocket and Jiffy are not suited for backend application storage systems as they provide no consistency guarantees for concurrent state updates. Shredder [50] reduces access latency to storage by executing functions on storage nodes. It is, however not scalable, with its current deployment being limited to only a single machine.

Another active area of development has been developing backend storage systems for persisting serverless application state. One such system is Anna [8], a key-value storage system that is elastic, and can scale across different tiers of storage based on application requirements. It uses lattice-based structures to merge concurrent updates and can guarantee causal consistency. It is used as the backend storage in the Cloudburst [27] serverless platform. Cloudburst addresses the latency overhead of using external storage by using the model of 'logical disaggregation and physical colocation'. It realizes this model by caching 'hot' application data on compute nodes. Caching can prevent external storage access, allowing state to be propagated between functions on the same machine more quickly. However, concurrent updates to data items at different cache sites can lead to inconsistencies. HydroCache [45] uses vector clocks to provide transactional causal consistency (TCC) guarantee for a set of functions that access data across multiple sites in Cloudburst. TCC provides causal+ consistency [32] for all I/Os in a transaction, even if they are issued from multiple sites. However, HydroCache would experience significant overhead when caches are frequently updated. Additionally, Cloudburst's consistency guarantees are unsuitable for applications requiring serializable execution.

Boki [25], an extension of Nightcore [26], is a serverless runtime designed for stateful applications. It is built on top of a shared log abstraction and introduces the concept of a 'metalog', allowing it to achieve strong consistency and fault tolerance. Boki can be used with DynamoDB as a logging layer to provide transactional guarantees and exactly-one semantics for serverless workflows (similar to Beldi [48]). However, it is up to 3.0x slower than natively using DynamoDB in this use case. It can also be expensive to read objects in Boki since it must replay the shared log to reconstruct the object's state.

We designed Arbor to serve as a storage system to persist state for serverless applications. Arbor tackles the latency traditionally associated with using an external storage system for this purpose by introducing an optimistic execution model. It buffers incoming transactions as a dependency graph and performs concurrency control checks asynchronously. It is developed to be scalable with its key-based partitioning of keys. Furthermore, developers can create policies for choosing which branch to commit by forking multiple branches of execution in response to conflicts rather than pessimistically aborting transactions using fixed policies.

2.2 Dependency Tracking Storage Systems

Transaction processing storage systems generally reconcile conflicts through policies that are indifferent to the specific semantics of the applications that are built on top of them. For example, systems that enforce serializability using optimistic concurrency control (OCC) [13] generally use the technique of timestamp-ordering to order transactions. In the case of conflicts, transactions with smaller timestamps assigned to them can commit while the others are aborted. In the case of COPS [32], a causally consistent geo-replicated storage system, cross-replica conflicts are resolved using a deterministic writer-wins policy. Crooks et al. [16] argue that storage systems should provide applications with a richer interface whereby they are exposed to conflicts and can implement their own policies to resolve them. For example, in the case of an e-commerce application built on top of a storage system, if there is a conflict between two customers purchasing the last stock of a given item, the application should be able to pick which customer's order is successful using its own policies such as selecting the customer with the larger cart, longer order history or the one with a premium subscription.

To this end, the authors introduce TARDiS [16], a dependency-tracking storage system that uses a novel branch-on-conflict concurrency control policy. TARDiS commits transactions synchronously, and conflicts result in the system creating a new logical 'fork' for the conflicting transactions. Each application client is given the view of sequential storage as it can work along only one branch at a time. TARDiS provides an interface that exposes branches to applications and allows developers to create application-specific merge functions to reconcile branches. Furthermore, it allows applications to specify the isolation level for each transaction, choosing from Serializability, Snapshot Isolation, or Read-Committed [12]. TARDiS consists of a multi-master replicated architecture and ensure causal consistency between replicas. It is intended for geo-distributed deployment. While TARDiS offers a similar user interface to Arbor, its replicated architecture makes it unsuitable for serverless computing, where a storage system must match the autoscaling nature of the compute. Moreover, it provides only causal consistency between replicas which would be insufficient for applications that require serializability.

ForkBase [43] is a multi-version key-value storage system for building 'forkable' applications such as blockchains and collaborative analytics. Its design choices are geared towards building blockchain-based applications, such as using a Structurally-Invariant Reusable Index for providing tamper evidence and data deduplication. Like TARDiS, a conflict in ForkBase results in a new fork for the object being created. However, forks are created at a per-object level, and there is no multi-object transactional support. Ficus [23] and Dynamo [17] also expose concurrent writes for users to resolve. However, this per-object level of branching is limited in its applicability, similar to the case of ForkBase.

Rococo [34] is designed to increase the amount of concurrency when processing transactions relative to standard protocols such as OCC and 2PL. A transaction is processed in two rounds. A coordinator first breaks each transaction into individual pieces and submits them to each participating server. The servers return dependency information that represents the relationship between other concurrently executing transactions that access the same data items. Next, the coordinator aggregates this dependency information from all servers and disseminates it to all the participating servers. Each server then reorders the pieces if they detect conflicts through the dependency information of the transaction and execute them. While this approach can increase throughput, it also adds latency to process each transaction. When contention is low, this can be a significant overhead.

2.3 Batch Transaction Processing

Batch processing is a technique used in OLTP systems to achieve higher throughput. Such systems buffer incoming transactions and process them in groups. During concurrency control, operations can be applied to the entire batch, making the computation more efficient. Usually, such systems trade off a long transaction completion time for high transaction processing throughput.

Ding et al. [19] reduces conflicts within OCC by collecting transactions in batches and reordering them both in the storage layer and during validation. Strife [38] is an in-memory database that collects transactions in batches and dynamically partitions most transactions into clusters that are executed in parallel without any concurrency control. It then executes transactions that cannot be partitioned by enforcing concurrency control.

Another use-case of batching is in the category of *deterministic databases*. Such databases usually consist of replicated architectures to ensure high availability. They prevent the need for expensive commit operations such as 2PC by typically totally ordering transactions before executing them in that order at each shard, consequently eliminating the need for concurrency control. Before executing, transactions in Calvin [41] acquire the required read/write locks in the order of Calvin's sequencer. Systems like BOHM [20] or PWV [21] achieve determinism by creating a dependency graph of the transactions in the batch. In such systems, it is vital to have prior knowledge of the read/write sets of transactions to construct the dependency graph and acquire the locks needed for ordering transactions. Aria [33] is a deterministic database that orders transactions only after they have been executed. However, it lacks the scalability required by serverless platforms since it does not partition data across machines.

Ding et al. and Strife are single-node implementations and do not meet the scalability requirements of serverless applications. Furthermore, clients of these systems, along with most deterministic databases, only submit transactions that the database reorders and executes in batches, typically as stored procedures. Instead, our goal with Arbor is to enable clients to issue interactive operations to the storage system and use the results optimistically without waiting for the storage system to validate their transactions. The systems described in this section are unsuitable for Arbor's use case since they do not support interactive client transactions.

Chapter 3

Optimistic Execution Model

This chapter presents Arbor's execution model. We begin by describing the model, its potential benefits, and how Arbor processes transactions in two rounds to enable this model. We then present Arbor's dependency-tracking client interface. Finally, we use an example to illustrate how the execution model works in practice.

3.1 Overview

Developers that compose serverless workflows typically use external storage systems to store application state [44, 25]. Platforms such as Beldi [48] and Boki [25] enable developers to build such application workflows with added transactional guarantees. These guarantees ensure that the application state remains consistent even if concurrent transactions make conflicting updates.

We designed Arbor, a backend transactional storage system for serverless workflows, to operate in an 'optimistic' model of client execution. In this model, functions within serverless workflows continue executing without waiting for transaction commits to backend storage to complete. Furthermore, the functions embed metadata within the transactions they submit, allowing the storage system to track the lineage of updates. This metadata identifies how data items depend on each other in terms of their read-write relationships.

The storage system uses this dependency information to organize the submitted transactions in a dependency graph logically. At this stage, the transactions are *graph-committed*. Their updates are globally visible to be read by other clients, but they can be aborted if the system finds them to violate serializability. The storage system checks if the constituent transactions of its dependency graph are serializable in a process called *finalization*. It finalizes transactions asynchronously, off the critical path of client execution by reorganizing the graph into a tree, where each branch represents a serializable ordering of transactions. The system forks new branches in the tree if it detects conflicting transactions. To arrive at a final state of the application data, it must pick one of the branches in the tree to commit and prune the rest. It can perform this branch selection based on custom policies defined by application developers. Furthermore, it can re-execute transactions that lie along pruned branches. The graph-commit and finalization steps collectively comprise a two-round model of committing transactions.

The optimistic execution model has three potential advantages. Firstly, it enables application clients to run quickly without added overheads due to the storage system enforcing concurrency control or executing a distributed commit protocol. This is because the storage system does not block clients from continuing their execution while it validates that each transaction is serializable. Consequently, this model can be particularly effective when workload contention is low since most transactions are likely to succeed in committing. Secondly, asynchronously performing concurrency control permits batch processing of transactions. This allows the storage system to process transactions more efficiently by applying computationally expensive operations to a whole batch of transactions, resulting in a lower overall cost. Thirdly, allowing branches to form rather than preemptively aborting conflicting transactions would give the system more context when deciding which branch to commit. Consequently, application developers can use this added context to implement dynamic branch selection policies. For example, in the case of an e-commerce application, if two customers order a given item at the same time, the application can decide which branch to commit based on which customer has the larger cart or a longer order history [16].

3.2 Client Interface

Arbor provides a multi-versioned key-value interface with transactional semantics as shown in Figure 3.1. Our modification to the *read* and *write* interfaces allows clients to embed dependencies that capture read-write relationships between data items.

A client can execute a transaction by calling *Begin* from which it will receive a transaction object. The client can then read values using the *Read* function. The client must include the key it wishes to read along with a policy which acts as a constraint on which version of the key to return. If a value is found, the library returns it along with its unique identifier. The default policy is for the client to read the latest version of the key.

Client Library
Begin() $\rightarrow$ *Transaction Object*
Transaction Object
Read(key, policy) $\rightarrow$ *(value, id)*
Write(key, val, dependencies) $\rightarrow$ *id*
Graph-commit() $\rightarrow$ *OK*
Abort()

Figure 3.1: Client interface

Consequently, this policy results in clients reading data that might not yet be committed. When issuing writes using the *Write* function, the client can embed *dependencies* as metadata. The dependencies are a list of identifiers representing the data used to compute the corresponding write. When the client is done executing the transaction, it can call *Graph-commit* to submit the transaction to Arbor.

3.3 Transaction Processing in Practice

Figure 3.2 illustrates an example of how Arbor processes transactions. First, during execution, clients create new versions of objects and embed dependency information into the transactions they graph-commit to Arbor. Next, using the dependency information, Arbor can organize the transactions using a dependency graph. After that, the finalization process begins. The first step is reorganizing the dependency graph into a tree where each branch represents a serializable execution order. We observe that concurrent updates to the key A result in multiple branches being forked. The second step is to choose one of the branches to commit and prune the rest. In this example, we choose the longer branch to minimize the number of re-executed transactions. Transactions along pruned branches are then re-executed until we are left with a single execution branch representing the order in which the corresponding updates are applied.

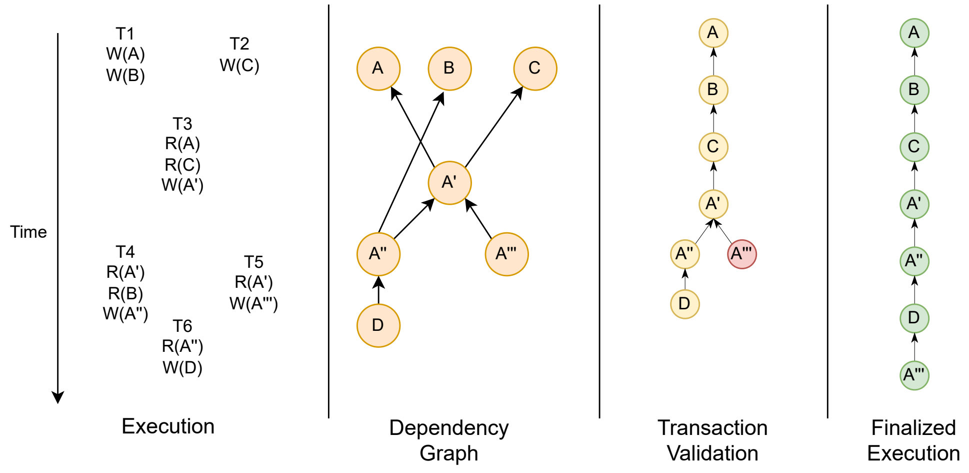


Figure 3.2: Example of transaction processing in Arbor. We use the ' symbol in the figure to distinguish different versions of an object. For example, we label the first version of key *A*. But when this object is updated, we label the new version as *A'*.

Chapter 4

Design

Arbor is a dependency-tracking storage system that operates in the same data center as its application clients. It partitions data by key across different shards, as shown in Figure 4.1. This allows it to scale to client requests while balancing load among its shards. Each shard is, in turn, composed of four major components: a data layer, an indexing layer, a communication layer, and a transaction processing engine.

Application clients that submit transactions to Arbor embed dependency information within the corresponding updates they make. This information captures the read-write relationships between data items. For example, the update corresponding to adding a new post to a user's timeline in a social media application will *depend* on the previous version of the user's timeline being updated and the data item corresponding to the post itself. The client can capture this relationship by including the identifiers associated with the two dependencies as arguments to the corresponding update they issue to Arbor. As we discussed in Section 3.1, this dependency-tracking interface has three main benefits: (1) it facilitates Arbor's two-round commit model where it can add transactions that the client has executed to a dependency graph quickly, thereby allowing clients to continue their execution without waiting for Arbor to acquire locks or validate read/write sets, (2) it allows Arbor to process transactions in batches, which can help reduce the overall cost of processing transactions and (3) it allows Arbor to create multiple branches of execution when validating transactions, and consequently, allow developers to implement policies for deciding which transactions to commit with added context.

Arbor utilizes the dependency information supplied by clients and stores data items as a dependency graph in the *data layer* of each shard. Within the graph, each vertex represents a data item, and edges represent dependencies between them. Because the data

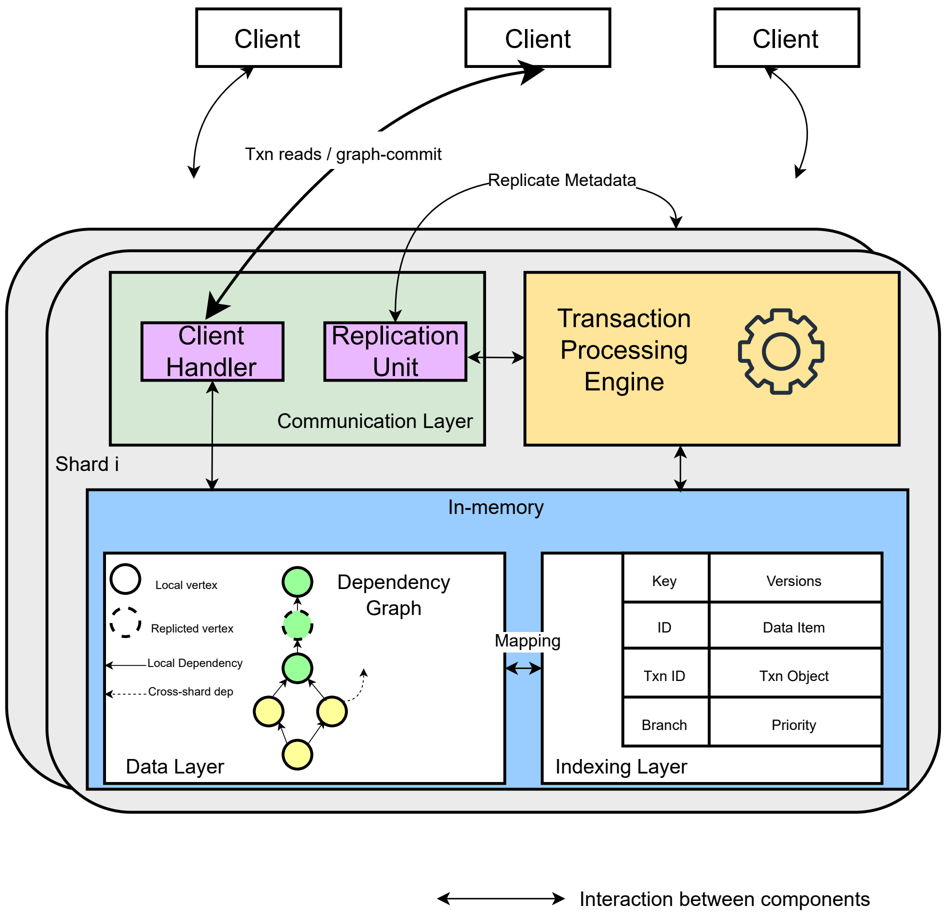


Figure 4.1: System Architecture

is partitioned, each shard holds the data of only a disjoint subset of the dependency graph. Edges within the graph are directed from child vertices to parent vertices to account for cross-shard dependencies. These unidirectional edges allow Arbor to record the dependency information of vertices even if their parent nodes are not located on the same shard. To finalize multi-shard transactions, the shards must all have the same global view of the dependency graph to all arrive at the same validation tree. To this end, Arbor uses a unique replication mechanism when processing transactions, replicating the dependency graph metadata (i.e., keys and edges) of transactions within the batch being finalized to all shards. Arbor processes each batch of transactions in isolation, so any incoming transactions added to the dependency graph do not affect those being processed.

Directly looking up data in Arbor's dependency graph would be inefficient since it would require traversing the graph to find the relevant data. Furthermore, when servicing read requests from functions within a workflow, Arbor should be able to return items that lie on the same chain of dependencies as the function's ancestors' updates. If a function uses versions of data from a different chain of dependencies, it can lead to non-serializable computation. To this end, Arbor indexes data across multiple attributes and holds these mappings in the *Indexing Layer*. As shown in Figure 4.2, include mapping transaction objects to their constituent writes, keys to their corresponding versions, data identifiers to their associated vertex in the graph, and keeping track of validated branches of execution during the finalization process. The list of validated branches is sorted based on a customizable priority, which developers can implement (see Section 4.4). In turn, Arbor uses this list primarily for two reasons: (1) when selecting a branch to append a transaction to, it chooses the branch with the highest priority that contains all the transaction's dependencies, and (2) after processing transactions in a batch, it finalizes the branch with the highest priority and prunes the rest.

Both the data layer and the indexing layer are stored in memory. Compared to secondary storage, this allows fast access to data. However, due to the volatile nature of memory, this design is not suitable for handling crashes or node failures. If data is too large to store in memory, it can also limit the applications that can be supported. However, data center machines typically have large memory sizes; for example, AWS EC2 virtual machines can be provisioned with up to 192 GB of memory [25].

The *communication layer* is composed of a client handler and a replication unit. The client handler serves client reads and graph-commits. When serving data reads, it uses the key-to-version mapping within the indexing layer to return a version of data that satisfies the client's read policy. When handling graph-commits, it adds the corresponding transaction writes to the dependency graph and populates the tables in the indexing layer. It then submits the transaction to the transaction processing engine, which finalizes trans-

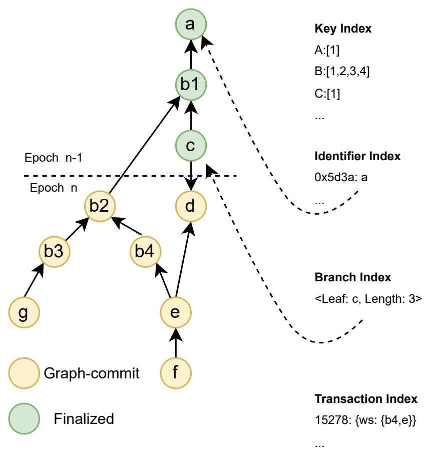


Figure 4.2: Illustration of the different mappings stored in each shard's indexing layer.

actions off the critical path of execution for clients. Based on input from the transaction processing engine, the replication unit broadcasts sections of the local dependency graph to other shards in the deployment. Additionally, it aggregates broadcasts received from other shards and propagates them to the transaction processing engine.

Each shard's transaction processing engine (TPE) finalizes transactions that have been graph-committed to the system in batches. Internally, the TPE has multiple stages (see Section 4.1) to process transactions, and all shards work in lockstep when processing each batch. When processing each batch, each TPE first independently finalizes the single-shard transactions within the batch. The TPEs then go through a *broadcast and aggregate* stage where the metadata of the dependency graph of the current batch of transactions is replicated to all shards. All the TPEs then finalize multi-shard transactions in the batch. Since the dependency graph is replicated before the TPEs process multi-shard transactions, and the finalization process is deterministic, the TPEs arrive at the same state of execution without the need for distributed commit protocols such as 2PC (See Theorem 4.3.1 and its proof).

4.1 Workflow of Transaction Processing

Figure 4.3 illustrates the transaction processing workflow in Arbor. When the client handler receives graph-committed transactions, it populates the corresponding entries in the indexing layer and adds the transaction's writes to the dependency graph in the data layer.

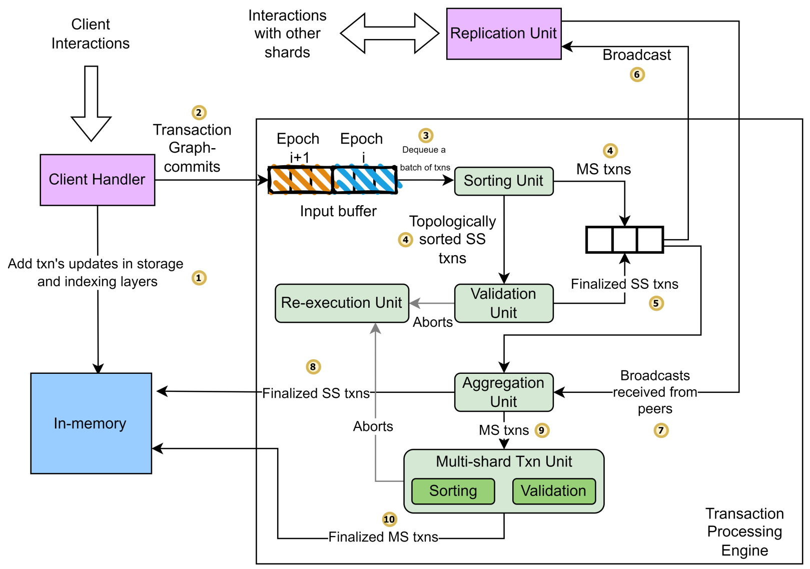


Figure 4.3: Workflow of transaction processing in Arbor.

It then adds the transaction to the TPE's *input buffer* (① and ② in Figure 4.3). The buffer enables a producer-consumer paradigm of execution, where the TPE can dequeue transactions that the client handler has appended. If the client handler appends transactions to the buffer at a rate faster than the TPE can handle, then the clients that issued those graph-commits are blocked until there is space in the buffer. This throttling mechanism is beneficial since it limits how much faster clients can graph-commit transactions relative to the rate at which Arbor can finalize them. Suppose the number of unprocessed transactions queued up grows significantly. In that case, it is more likely to suffer cascading aborts as there is a larger window of time in which conflicts can occur.

The TPEs of all the shards collectively process transactions in batches and work in lockstep. Each TPE reads a fixed size of transactions from the input buffer based on the *batch size* configured during initialization (③ in Figure 4.3). The TPE defaults to a timeout if insufficient transactions are available. Transactions that are read collectively by all TPEs form the batch that they will process.

The transactions then enter a sorting unit where they are seggregated into groups of single-shard and multi-shard transactions, respectively. Multi-shard transactions are placed in a buffer for processing in a later step. The sorting unit iterates through the group of single-shard transactions and aggregates the dependencies of their constituent writes. It sets this aggregated list as the dependency of the transaction as a whole. As a transaction's writes are only visible to other transactions after it has completed its execution, cyclic dependencies between transactions are not possible. It then sorts the single-shard transactions topologically [28] based on their dependencies (④ in Figure 4.3).

As part of its finalization process, Arbor reorganizes sections of its dependency graph to a validation tree where each branch represents a serialized execution, and conflicts result in new branches being forked. Once topologically sorted, the group of single-shard transactions enters a validation unit. Here, the TPE selects a branch from the validation tree containing all of the transaction's dependencies, and if it does not detect any conflicts, it adds the transaction to the end of the branch. However, if conflicts exist with the transaction being added to the branch, the validation unit forks a new branch and adds the transaction to it (see Section 4.3). After adding all the transactions in the group to the validation tree, it selects one of the branches to finalize based on their respective priorities. The validation unit prunes other branches and propagates the corresponding transactions to the re-execution unit asynchronously.

The group of finalized single-shard transactions are then added to the same unordered buffer in which the multi-shard transactions were added to (⑤ in Figure 4.3). The TPE then propagates the data in the buffer to the replication unit (⑥ in Figure 4.3). The TPE only serializes the required metadata (transaction ID, keys and dependencies from the transction's write set) from this buffer and not the actual data values. In turn, the replication unit broadcasts this data to all the other shards in the deployment. The TPE is then blocked until it receives broadcasts from all other shards through the replication unit (⑦ in Figure 4.3). This stage enables the shards to work in lockstep when processing each batch, as they all must complete finalizing single-shard transactions before continuing with their execution.

Once the TPE has received broadcasts from all its peers, it populates its dependency graph in the data layer and the mappings in the indexing layer with the corresponding data (⑧ in Figure 4.3). Since each shard finalizes multi-shard transactions without coordinating with other shards, they must all be operating on the same state of data before proceeding. To achieve this, all the shards deterministically aggregate the received branches of finalized single-shard transactions into a single branch which represents a total order of execution of these transactions. This is illustrated in Figure 4.4. Each TPE links the branches of single-shard transactions together in an order that is configured when the system is initialized.

The aggregation unit then propagates the unprocessed multi-shard transactions received by all the shards in the current batch to the multi-shard transaction unit. Here, they follow the same steps as single-shard transactions for finalization. They are first

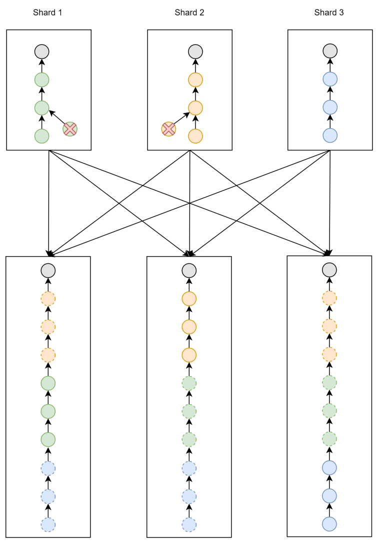


Figure 4.4: Illustration of how each Arbor shard deterministically aggregates the execution order of single-shard transactions finalized independently at different shards. We use different colors to distinguish transactions finalized at different shards. Transactions that are crossed out in the figure represent those that are pruned during finalization.

topologically sorted and then subsequently added to a tree following a validation process (see Section 4.3). The multi-shard transaction unit then adds the branch of finalized transactions to the data layer. Any transactions that are aborted during this stage are re-executed asynchronously. To prevent duplicate re-executions, only the shard at which an aborted multi-shard transaction was graph-committed to will trigger the re-execution of the transaction.

The process of finalizing transactions is an intrinsically serial one since Arbor must validate each batch of transactions before it can add transactions from a future batch to the validation tree. However, the architecture of the TPE allows some level of concurrency between batches. Batches of transactions that will be processed in the future can be segregated and topologically sorted (④ in Figure 4.3) concurrently while the current batch of transactions is being finalized (⑤ - ⑩ in Figure 4.3).

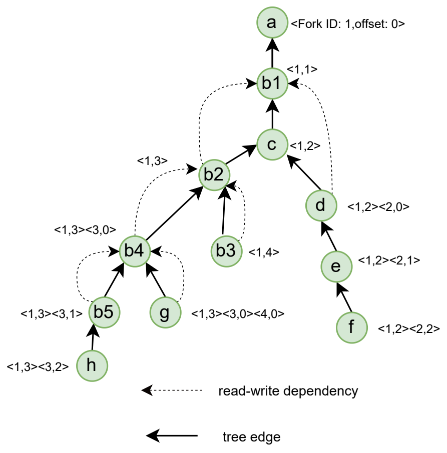


Figure 4.5: Example showing the lineage addresses of the vertices in a finalized tree.

4.2 Lineage Addressing

During the finalization process, when Arbor reorganizes its dependency graph into a tree, there is a requirement to efficiently identify if two given nodes lie on the same path in the tree. This is because: (1) when adding a vertex to a branch, we must verify that all its parent dependencies lie on that branch and (2) we must check if there are any conflicting updates with the vertex's transaction. To meet this need, we developed an addressing mechanism called *lineage addressing*, inspired by TARDiS's *fork paths* [16]. We will first describe how the lineage addressing scheme works and then describe how it is different from the scheme used in TARDiS.

Figure 4.5 illustrates an example of a tree created during the finalization process and how lineage addresses are assigned to vertices. Each lineage address consists of a list of (fork ID, offset) pairs. The lineage address of a vertex concisely represents the position of the vertex relative to the root node. The path from the vertex to the root may contain multiple fork points. The vertices between any two fork points constitute a 'segment' and the entries in a vertex's lineage address represent each of the segments in the path from the vertex to the root. Consider the vertex f in Figure 4.5. There are two segments corresponding to this node. One from node a to node c and the other from node d to node f. These are captured by the entries in its lineage address. The first entry (1,2) represents the section of the path until the first fork point, node c. The second entry (2,2) represents the second section of its path, ending at the vertex f itself. When a new branch is forked, the addresses of the existing branch do not change. This is why the addresses of nodes *b2* and *b3* do not have new entries added even though other branches were forked.

Algorithm 1 Branch Address Comparison

```
1: function COMPARELINEAGE($a$, $b$)        ▷ $a$ and $b$ are both Lineage Addresses
2:     if len($a$) == len($b$) then
3:         $idx$ = len($a$) - 1
4:         return $a[idx]$.ForkId == $b[idx]$.ForkId
5:     end if
6:     var $shorter$, $longer$ LineageId
7:     if len($a$) >len($b$) then
8:         $idx$ = len($b$) - 1
9:         $shorter = b[idx]$
10:        $longer = a[idx]$
11:    else
12:        $idx$ = len($a$) - 1
13:        $shorter = a[idx]$
14:        $longer = b[idx]$
15:    end if
16:    return ($shorter$.ForkId == $longer$.ForkId) and ($shorter$.Depth <= $longer$.Depth)
17: end function
```

Algorithm 1 illustrates how lineage addresses can be compared in constant time to identify if two vertices lie on the same branch. If the lineage addresses of two vertices have the same number of entries and their Fork ID's match, then they are on the same branch (lines 2-5). In cases where they have different numbers of entries, we only compare the entry at the index that corresponds to the last element of the shorter lineage address. When the Fork IDs match and the entry in the shorter lineage address has a smaller depth than the corresponding entry in the longer lineage address, we can infer the two vertices are on the same branch.

Unlike Arbor, which can fork new branches from the middle of existing ones, TARDiS only forks new branches as siblings to existing leaf nodes in its state tree. TARDiS's fork path scheme comprises a list of (i, b) pairs. Each entry in a given vertex's fork path represents that the vertex is a descendant of the b^{th} child of the state i [16]. As a result, when a new branch is forked from a given vertex for the first time, the fork path of the existing branch must be updated with a new entry. If Arbor used this scheme as is, it would require a recursive update of fork paths of all vertices present on the existing branch that the new one is being forked from, thereby potentially adding significant overhead to the fork operation. Instead, the semantics of using (fork ID, offset) pairs precludes the need for such a recursive update, making forking new branches a computationally cheap operation.

4.3 Concurrency Control

We now describe how Arbor's transaction processing engine (Figure 4.3) validates whether transactions within a batch are serializable. Figure 4.6 presents an overview of the process. Arbor first picks a batch of graph-committed transactions. Rather than recursively traversing through its dependency graph, Arbor selects transactions from the TPE's input buffer (as we discussed in Section 4.1). Next, it aggregates each transaction's dependencies and sorts them in topological order using the dependencies. Arbor then iterates through the topologically sorted list of transactions and adds them to a branch of a validation tree if they pass the required checks. If conflicts exist, Arbor forks a new branch and appends the transaction to it. Each branch in the validation tree represents a serialized order of transactions and edges between vertices reflect their ordering rather than read-write dependencies. In cases where creating a branch is impossible, Arbor aborts and re-executes the transaction. After adding all the transactions to the validation tree, Arbor picks one of the branches to commit and prunes the remaining branches. It re-executes the corresponding transactions and ends up with a single branch of transactions that reflects the serialized order of execution.

Algorithm 2 illustrates how transactions are added to a branch. This process involves determining the branch with the highest priority that contains all the transaction's dependencies, checking for conflicts, and either forking a new branch or adding the transaction to the selected branch. We begin by iterating through each transaction within the topologically sorted list and check if any of its dependent transactions have been aborted (lines 5-8). If so, we abort the current transaction and move on to the next one. When multiple branches of execution exist, we check if all of the transaction's dependencies lie on the same branch and return the longest such branch if the condition is satisfied (lines 10-13). If not, we abort the current transaction and move to the next one on the list. This is because each branch represents a distinct serialized execution. If a transaction depends on data from different branches, it cannot be placed on any single one.

Given a candidate branch from the previous step, we check if there are any conflicting updates with the current transaction that we wish to add to the branch using the conflict window technique (lines 16-21). Finally, we append the transaction to the candidate branch if we detected no conflicts. In case of conflicts, we create a new branch from the

Algorithm 2 Concurrency Control

```
1: $abortList$ = []
2: function CONCURRENCYCONTROL($txnList$)    ▷ $txnList$ is a topologically sorted list of transactions
3:     for $txn$ in $txnList$ do
4:         if $checkAbort(txn.dependencies)$ then
5:             $txn.Abort$ = True
6:             $abortList.append(txn)$
7:             continue
8:         end if
9:         $branch$, $branchIdx$, $abort$ = $checkSameBranch(txn.dependencies)$
10:        if $abort$ == True then
11:            $txn.Abort$ = True
12:            $abortList.append(txn)$
13:            continue
14:        end if
15:        $conflictPoint$, $abort$ = $checkConflict(txn, branch)$
16:        if $abort$ == True then
17:            $txn.Abort$ = True
18:            $abortList.append(txn)$
19:            continue
20:        end if
21:        if $conflictPoint$ != None then
22:            $branch$ = $createBranch(conflictPoint.Parent)$
23:            $branchList.append(branch)$
24:            $branchIdx$ = $len(branchList$ - 1)
25:        end if
26:        $setLineage(txn)$
27:        $addTxn(txn, branch)$
28:        $branchList.MaintainPriority(branchIdx)$
29:    end for
30:    $abortList.append(pruneBranches())$
31:    $ReExecute(abortList)$
32: end function
```

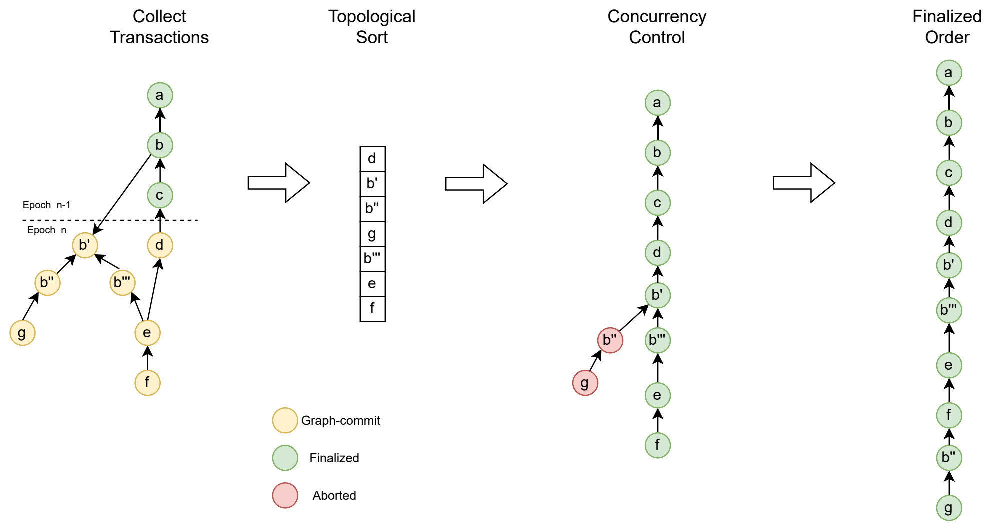


Figure 4.6: Data-oriented overview of the transaction finalization process

parent vertex of the earliest conflicting update and add the transaction to that branch. Once a transaction is added to a branch, we must update the branch's priority and its position within the sorted list of branches (lines 22-26). We update the priority using the policy implemented by the application developer (described in Section 4.5). Moving the augmented branch to a new position in the list is an $O(n)$ operation where n is the number of branches, since the rest of the list is sorted correctly.

To detect conflicts within a branch, we use the notion of a 'conflict window' from Tango [10] for each of the transaction's dependencies. Unlike Tango, where a conflict window is defined using $BeginTX$ and $EndTX$ records to define when the client is executing a given transaction, we instead define a conflict window for each of the transaction's dependencies as the section of the selected branch starting from the leaf node till the dependency itself. Within this window, if there are any new versions of the key corresponding to the dependent vertex, then a conflict exists. This is because if we add the transaction as a new leaf node to such a branch, the newer version of the dependent key will make the data used by the transaction stale, so adding the transaction to the end of such a branch will not result in serializable execution.

Consider Figure 4.7 where the transaction being added to the branch has two dependencies, i.e, a and b'. We define two conflict windows, one for each dependency. The conflict window for key b has no conflicting updates. On the other hand, there is a conflicting update in the conflict window of key a. As a result, we cannot add the transaction at the

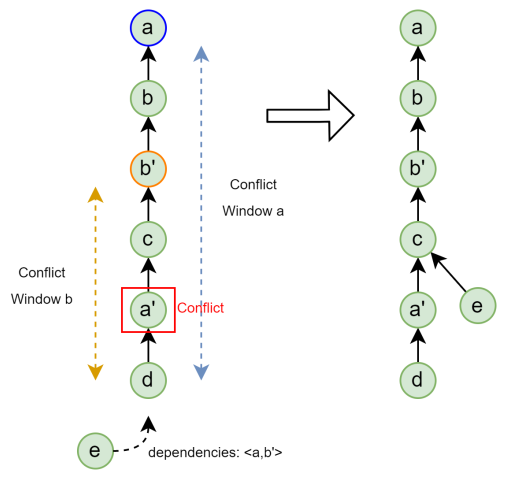


Figure 4.7: Illustration of how Arbor uses the concept of conflict windows to detect conflicts. It defines a conflict window for each dependency of a transaction and checks if any new versions of the corresponding key exist in each corresponding window. In the figure, we see that a conflict exists for the dependency to key *a* as a conflict exists within the window.

end of the branch and instead fork a new one at the parent vertex of the conflict. The last step of the finalization process is for Arbor to select the highest priority branch created during the transaction validation step and prune the rest.

4.3.1 Correctness of Distributed Transaction Processing

Theorem 4.3.1. *After processing each batch of transactions, all shards arrive at the same finalized branch.*

Proof. (BY CONTRADICTION.) Let us assume, on the contrary, that the shards arrive at different finalized branches after processing a given batch. There can be two causes for this: (1) they have ordered single-shard transactions differently, or (2) they created different validation trees when finalizating multi-shard transactions.

Each shard independently finalizes the single-shard transactions it is responsible for within a batch. As shown in Figure 4.4, the shards then link the aggregated branches together in an order defined when the system is initialized. Therefore, the shards cannot arrive at different orders for the single-shard transactions within the batch.

```
type Priority  interface  {
    AugmentPriority(leaf Priority,  txnPriorityMetadata interface{})
    ComparePriority(branchA, branchB  Priority)
}
```

Figure 4.8: Customizable Branch Selection Policy Interface

Multi-shard transactions within the batch are replicated across all shards and finalized by all. For the shards to create different validation trees when processing these transactions, they must have either arrived at different topological orders or made different decisions when detecting conflicts. Each shard first sorts the multi-shard transaction in ascending order of their associated transaction identifiers and then uses Kahn's algorithm [28] to order them topologically. Sorting the transactions by their identifiers first ensures that in Kahn's algorithm, transactions that are logically siblings to each other in their dependency relationship are ordered the same across all shards. Consequently, this ensures that all the shards arrive at the same topological order for multi-shard transactions. When validating multi-shard transactions, they are added to the same finalized tree across all shards and use Algorithm 2 to detect conflicts, which is deterministic. As a result, it is impossible for different shards to make decisions when adding multi-shard transactions to their validation tree.

Hence, our assumption was incorrect, and all shards arrive at the same finalized branch after processing each batch of transactions. □

4.4 Branch Selection

For developers to implement custom branch selection policies, we implemented the interface shown in Figure 4.8. The interface consists of two functions, namely *AugmentPriority* and *ComparePriority*. This interface gives developers the freedom to implement the logic to define a branch's priority based on the semantics of their applications. As clients execute transactions, they can embed metadata for Arbor to use when invoking the branch selection policy. During finalization, Arbor invokes the developer's *AugmentPriority* function that can utilize this metadata to update the corresponding branch's priority. After adding the transaction to a branch, Arbor maintains its sorted list of branches using *ComparePriority*.

Consider Figure 4.9, which illustrates an implementation of a policy for selecting the longest branch. When clients are done executing each transaction, they can embed the length of the transaction's write set as *priorityMetadata*. When a transaction is added

```
type LongestBranchPriority struct {
    length int64
}

func NewLongestBranchPriority() *LongestBranchPriority {
    return &LongestBranchPriority{length: 0}
}

func (lp *LongestBranchPriority) AugmentPriority(leaf Priority, txnPriorityMetadata interface{}) {
    parentPriority := leaf.(*LongestBranchPriority)
    writeSetCount := txnPriorityMetadata.(int64)
    lp.length = parentPriority.length + writeSetCount
}

func ComparePriority(branchA, branchB Priority) bool {
    A := branchA.(*LongestBranchPriority)
    B := branchB.(*LongestBranchPriority)
    return A.length > B.length
}
```

Figure 4.9: Implemention of Longest Branch Selection Policy

to a branch, Arbor invokes the *AugmentPriority* function, which increments the branch's priority by the length of the transaction's write set. When maintaining the sorted order of branches (as described in Algorithm 2, Arbor utilizes the *ComparePriority* function, which sorts the branches in descending order of their length. This interface is simple yet powerful because it allows developers to define a branch's priority based on its constituent transactions' characteristics.

4.5 Implementation

We implemented our prototype of Arbor in the Go programming language. The codebase consists of approximately 5.2K lines of code. We implemented a custom TCP socket-based RPC library for I/O between clients and servers as we found that gRPC was not able to saturate 10 Gb/s network links. Rather than spawning a new Go routine on the server-side for every RPC invocation, our custom RPC library establishes persistent connections between clients and servers during the experiment. We used the msgp [6] library to marshal and unmarshal data for communication over the network.

Our prototype of Arbor uses Go map objects to store the mappings of the indexing layer. We implement a custom interface on top of standard map objects which shard data across multiple maps for increased concurrency, and use a read/write mutex with each individual map to prevent inconsistencies arising out of race conditions. Furthermore, we maintain separate copies of the maps, one for serving client requests and the other for the transaction processing engine. This decouples client requests from blocking background transaction processing. We add data to the the copy that we use to serve client requests when clients graph-commit transactions. On the other hand, we add data to the copy used for background processing during the sorting and aggregation units of the TPE.

We implemented 4 read policies in our prototype of Arbor: (1) *Latest*: read the latest submitted version of a key that can correspond to either a graph-committed or finalized transaction, (2) *Finalized*: return the latest submitted version of a key that has been finalized, (3) *Versioned*: return the version of the key corresponding to a version specified as an additional argument and (4) *Tagged*: read the latest client submitted version of a key corresponding to a user-defined tag specified as an additional argument.

Chapter 5

Evaluation

In this chapter, we present the results of our experiments comparing Arbor against related systems using various deployment configurations and workload access patterns. We begin by describing the experimental setup in Section 5.1 and the metrics we use to gauge Arbor's performance in Section 5.2. We then describe the workloads we use in Section 5.3 and the evaluated systems in Section 5.4. Finally, we present our results in Section 5.5.

5.1 Experimental Setup

We conducted our experiments using nine machines from a local cluster. Each machine has a 12-core Intel Xeon E5-2620 CPU that runs at 2.10 GHz with 64 GB of RAM and Ubuntu 20.04.3 LTS with kernel 5.11.0-34-generic. 10 Gb/s network links connect the machines. We designated six machines for running clients in our experiments. We configured each machine to run with at most ten concurrent transaction-issuing client threads to preclude any performance bottlenecks from arising during the experiment.

We configure Arbor to use a batch size of 5000 transactions. However, if an insufficient number of transactions have been graph-committed to trigger transaction processing, we default to a timeout of 300 ms from when Arbor began collecting transactions for the current batch. We disable re-execution of aborted transactions. We run each experiment for 140 seconds and exclude results from the experiments' first 20 seconds to achieve steady state. We continue to issue transactions from clients until all transactions from the 120-second steady state period have been processed and end the experiment at this point. This is done to keep each system in a steady state throughout the experiment. We repeated each experiment three times, and the data points in our figures represent the mean value from all repetitions, while error bars represent the 95% confidence intervals.

For executing workloads, we initially used Nightcore [26], a serverless platform optimized for latency-sensitive applications. To evaluate its performance, we deployed it on a local cluster of 4 machines and observed a peak throughput of only 17,000 invocations/second when invoking a function that returns "Hello World" to the invoking client. The limited throughput results from each function invocation having to propagate through a gateway, dispatching queue, per-request tracking log, and, finally, the corresponding function container. As a result, if we used Nightcore, we would have insufficient load during our experiments to saturate the storage systems in our evaluation. We assume that when using a commercial serverless platform such as AWS Lambda [2] to deploy an application, the cloud provider can scale compute resources sufficiently to match the function invocation load.

5.2 Performance Metrics

Arbor's performance is measured using the following metrics. Note that aborted transactions are not included in any of the calculations.

- **Finalization Throughput** measures the rate at which Arbor finalizes submitted transactions. To determine the finalization throughput, we first measure the time between the beginning of the steady state (20 seconds after each experiment begins) and when all transactions submitted during the steady state have been processed. We then compute the throughput by dividing the number of finalized transactions by this time duration.
- **Finalization Latency** Finalization latency measures the average time it takes for a transaction to be finalized after it has been graph-committed. We record the timestamps at which each transaction is graph-committed and finalized. After each experiment, we calculate the finalization latency by finding the difference between the two timestamps for each transaction and taking the average for all transactions submitted during steady state.
- **Graph-commit Throughput** measures the rate at which transactions are graph-committed during the steady state of the experiment.
- **Graph-commit Latency** measures the time between when a client starts executing a transaction and when it has completed graph-committing it.

5.3 Benchmark Workloads

In our evaluation, we use the YCSB+T [18], and Retwis [31] workloads. YCSB+T is an extension of YCSB [14], a popular workload used to evaluate key-value stores, with multi-operation transactional semantics [49]. In most of our experiments that use the YCSB+T workload, transactions are primarily configured to read, modify, and write (RMW) two keys. With this multi-key RMW configuration, clients can embed dependency information within writes, which is how we expect applications that use Arbor to construct transactions. We pre-fill each system we evaluated with 10 million keys in our YCSB+T experiments as done in Carousel [46]. Each value is 1 kilobyte in size.

Retwis is a Twitter-like workload with many-to-many relationships between users. Our implementation of Retwis is similar to that of Augustus [37]. Each user has a follower list (list of followers), a following list (list of other users that this user follows), and a timeline consisting of posts from users they follow [37]. There is a maximum limit of 50 posts in each user's timeline (default value used in TARDiS [16]). The Retwis workload consists of the following transactions: *Follow*, *Post* and *Load Timeline*. The *Follow* transaction involves one user following another. It consists of two RMW operations with updates to one user's follower list, and the other user's following list [37]. The *Post* transaction involves creating a unique post for a given user and adding it to the timelines of the user's followers. Finally, the *Load Timeline* transaction involves reading all of the posts within a given user's timeline. The Retwis workload is a good candidate to evaluate Arbor since each transaction involves updates to many keys, and unlike YCSB+T, transactions involve dependency relationships between different objects. For example, in a *Post* transaction, when adding a post to a follower's timeline, the new version of the timeline depends on the older version that the client read and the post object.

We use two configurations of Retwis in our experiments that are the same as those used in the evaluation of TARDiS [16]. These include *Post-Heavy*, which consists of 65% Load operations, 30% post operations and 5% follow operations and *Load-Heavy* consisting of 85% load operations, 10% post operations and 5% follow operations. We used the default value in TAPIR [49] and Natto [47] and pre-fill each system we evaluated with 10 million users in our Retwis experiments.

5.4 Evaluated Systems

We do not compare Arbor against some of the related systems we described in Chapter 2, such as TARDiS [16] and Boki [25], as these systems store data on secondary storage. As a result, it would be unfair to compare these systems to Arbor. Instead, we evaluate Arbor against two other systems that are in-memory databases and enforce the serializability of transactions.

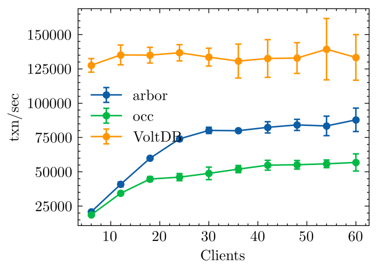
The first system we compare against is VoltDB [40]. It is a popular in-memory SQL database for production use. In contrast to Arbor, it does not support interactive operations within transactions since they are executed as stored procedures. As a result, the client and server only need to make one network round-trip to issue a transaction. VoltDB is optimized for workloads in which transactions either use keys belonging to only one partition of a table or transactions which it can break up into smaller units that it can independently execute at each partition. However, when executing other types of multi-partition transactions, VoltDB's transaction coordinator locks all partitions in the cluster, regardless of how many partitions are participating in the transaction [5].

The second system we compare against is a transactional key-value store that uses optimistic concurrency control (OCC) and the two-phase commit (2PC) protocol to commit cross-partition transactions. This system is derived from the implementation of the Spanner [15]-like system used in the evaluation of Natto [47]. This system commits transactions synchronously and it executes the second phase of 2PC lazily.

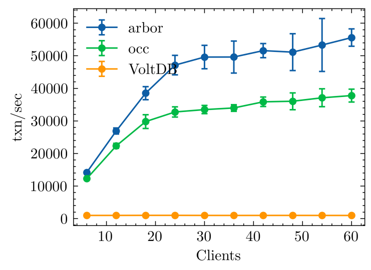
For a fair comparison, we implemented the OCC+2PC system in the Go language and configured it to use the same custom socket-based RPC library as Arbor. We used the Java client for VoltDB instead of the Go language client since we found it was significantly faster when using the same workloads. To maximize VoltDB's performance, we followed its documentation [7] and configured clients to issue transactions asynchronously and implemented callback functions to interpret if transactions are committed or aborted.

5.5 Results

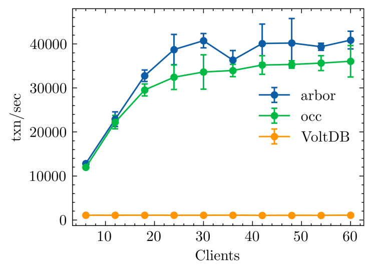
We present our results from experiments on a single-machine deployment in Section 5.5.1. This configuration eliminates the difference between single-shard and multi-shard transaction processing in Arbor. We then present how Arbor scales in throughput by adding more machines and partitioning data across them in Section 5.5.2, in a workload consisting of only single-shard transactions. We illustrate the effects of cross-partition transactions in Section 5.5.3 and skewed access patterns in Section 5.5.4. Finally, we show how Arbor's two-round commit model reduces execution time for applications that sequentially execute transactions in Section 5.5.5.



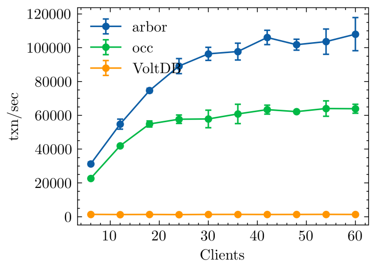
(a) Single Key RMW



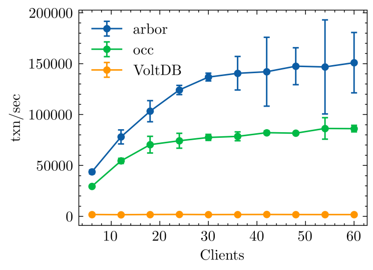
(b) Multi-key RMW, Uniform



(c) Multi-key RMW, Skew



(d) Retwis, Post-Heavy



(e) Retwis, Load-Heavy

Figure 5.1: A comparison of throughput achieved with Arbor, OCC, and VoltDB's on a single machine. We report finalization throughput for Arbor along with the transaction commit throughput for the other two systems.

5.5.1 Single machine deployment

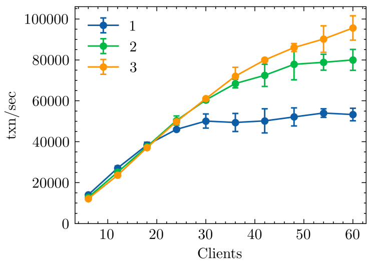
We begin our evaluation by considering how a single-machine deployment of each system performs under different access patterns.

Figure 5.1(a) illustrates results from experiments consisting of single-key RMW YCSB+T transactions. This configuration favors VoltDB because each transaction can be executed independently by each table partition, and consequently, there is a high degree of parallelism. As a result, VoltDB can achieve 1.57x the peak throughput of Arbor. Despite the optimizations in Arbor's batch-processing pipeline, since it does not process transactions within a batch in parallel, it cannot make up the deficit to VoltDB. Moreover, transactions executed by Arbor and the OCC+2PC system involve an extra network round-trip per read operation relative to VoltDB, where transactions are executed as stored procedures. We found that deploying more Arbor shards on the same machine did not improve performance, as the CPU is completely utilized even when using one shard. The benefits of Arbor's batch processing pipeline come to light when comparing Arbor to the OCC+2PC system, where Arbor achieves a speedup of 1.54x in peak throughput.

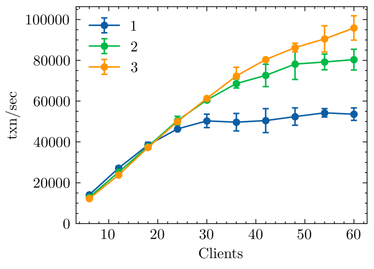
We next consider the multi-key RMW configuration of YCSB+T with a uniform key distribution (Figure 5.1(b)). In this case, VoltDB suffers from the coordination overhead it must enforce for transactions that involve multiple table partitions. As we mentioned before, VoltDB's transaction coordinator locks all partitions in the cluster to commit such transactions, regardless of how many partitions are participating in the transaction [5]. As a result, we observe a sharp drop in throughput for VoltDB relative to the single-key RMW case, and the resultant peak throughput is 1003 transactions per second (tps). Arbor's peak throughput drops by 36.8% from 87.9K tps to 55.6K tps. This is because each transaction has twice the number of dependencies as the previous case, which increases the time taken to add transactions to the validation tree, as Arbor creates a conflict window for each dependency of a transaction. The peak throughput of the OCC+2PC system falls by 15.3%. The relatively modest decrease is because the major bottleneck for this system is the synchronous processing of transactions. For this access pattern, Arbor achieves a speedup of 55.4x and 1.47x relative to VoltDB and the OCC+2PC system.

In the case of a skewed key distribution (Zipfian coefficient = 0.8) as shown in Figure 5.1(c), we observe that Arbor's peak throughput falls to 40.8K tps. The primary cause for this is the increase in abort rates to 24.5% when Arbor reaches its peak throughput. In the case of OCC, the abort rate goes up to only 1.9%. Arbor's effective time to commit a transaction ranges from 200-650 ms, whereas the OCC systems's is up to 1.6 ms. This means that the effective window for writes to become stale is much higher in the case of Arbor and results in a much more significant rise in abort rates and, consequently, a drop in throughput.

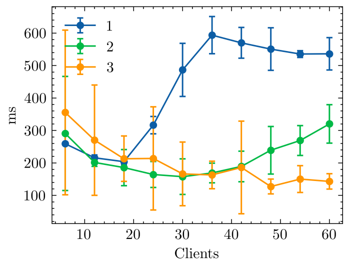
Finally, we consider the Retwis workload illustrated in Figures 5.1(d) and 5.1(e). Since both configurations consist primarily of read-only transactions, the absolute throughput is significantly higher for the Retwis workloads than for the RMW YCSB+T transactions. However, as in the case of the YCSB+T experiments, we observe that Arbor has a speedup in peak throughput relative to the OCC+2PC system in the post-heavy and load-heavy configurations by 1.69x and 1.75x, respectively. VoltDB's performance is similar to the multi-key YCSB+T experiments. This is because all of the transactions in the Retwis workload involve data from multiple partitions of the consistituent tables. Due to this, VoltDB processes transactions relatively slowly since all table partitions have to be locked when processing each transaction.



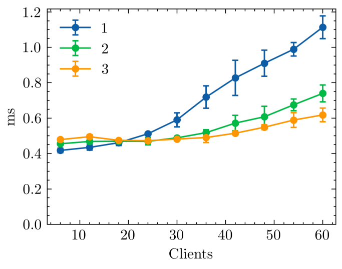
(a) Finalization Throughput



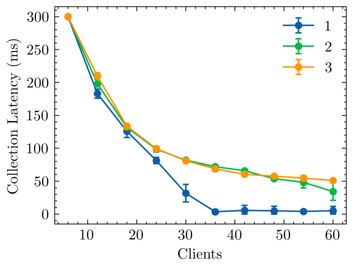
(b) Graph-commit Throughput



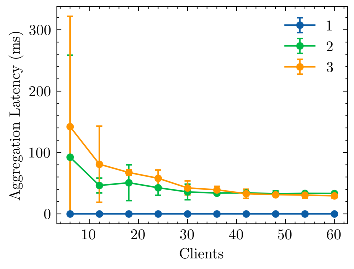
(c) Finalization Latency



(d) Graph-commit Latency

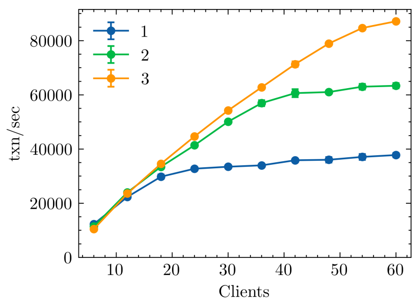


(e) Collection Latency

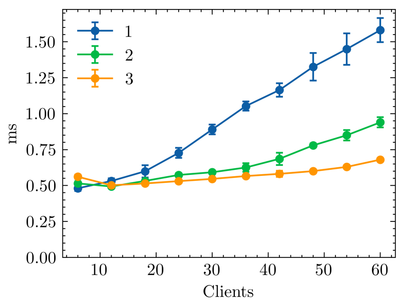


(f) Aggregation Latency

Figure 5.2: Analyzing Arbor's scalability using a YCSB+T workload consisting of single-shard, multi-key RMW transactions with a uniform key distribution.



(a) Throughput



(b) Latency

Figure 5.3: Analyzing the OCC+2PC system's scalability using a YCSB+T workload consisting of single-shard, multi-key RMW transactions with a uniform key distribution.

5.5.2 Scalability

We next evaluate Arbor's scalability by observing how performance changes based on the number of shards used in the deployment. In the results illustrated in Figure 5.2, we deploy each shard on different machines and use a multi-key RMW workload consisting of only single-shard transactions (we will gauge the effect of multi-shard transactions in Section 5.5.3). We adjust the batch size used in each set of experiments so that the collective batch size of all shards is the same. We use a batch size per shard of 5000 transactions when deploying one shard, 2500 transactions using two shards, and 1666 transactions using three shards. Due to the limited number of client machines available, we could not saturate finalization latency in the two-shard deployment and finalization throughput in the three-shard deployment.

We observe peak finalization throughput scales from 54K to 80K to 95.6K tps as we add additional shards, as seen in Figure 5.2(a). Adding more shards enabled more concurrent transaction processing as each shard can independently finalize submitted transactions. There is, however, an associated cost with broadcasting and aggregating the finalized branches across all shards when processing each batch of transactions. Figure 5.2(f) illustrates this latency which converges to 33.3 ms in the case of 2 shards and 29.3 ms in the case of 3 shards. The smaller batch size of each shard in the latter case explains the lower convergence value. As a result of this overhead, the peak throughput does not scale linearly with the number of shards. Additionally, Figure 5.2(d) shows that adding more shards allows Arbor to serve more clients with graph-commit latency staying below 0.8 ms for significantly more clients.

Figure 5.2(c) illustrates that finalization latency first decreases as the number of clients increases. The reason is that when there are few clients, the transaction graph-commit rate from clients is the limiting factor. Figure 5.2(e) corroborates this explanation, which shows the average time taken to collect a batch of transactions from the graph-commit queue on the server side. We observe that the finalization latency for the single-shard deployment begins to plateau at 36 clients. We throttle clients from executing if the size of a shard's input queue exceeds two batch sizes. This mechanism prevents finalization latency from becoming too large to the point where cascading aborts become significant. This throttling is why the finalization latency plateaus rather than exponentially rising.

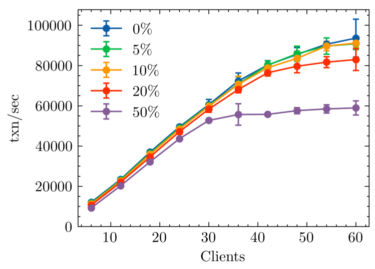
Figure 5.3 illustrates the scalability of the OCC+2PC system using the same workload as the Arbor experiments. The peak throughput scales from 37.8K tps with one shard to 63.4K tps and 87.1K tps with two and three shards, respectively. Since the workload consists of only single-shard transactions, there is no coordination between shards required to process transactions. As a result, the relative scalability is higher than that of Arbor. However, the peak throughput values in each configuration is lower than Arbor.

5.5.3 Cross-Partition Transactions

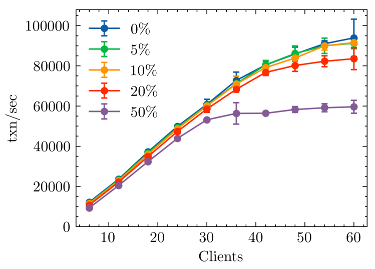
We now consider the effect of cross-partition transactions on the performance of Arbor and the OCC+2PC system. We deploy each system using three shards, each on different machines. We use the multi-key RMW YCSB+T workload and vary the portion of cross-partition transactions in the workload. Figures 5.4 and 5.5 illustrate the results of our experiments. We observe that in both Arbor and the OCC+2PC system, the percentage of cross-shard transactions increases, throughput decreases, and latency increases. Compared to the OCC+2PC system, Arbor achieves a relatively higher finalization throughput for all workload configurations with less than or equal to 20% of cross-partition transactions. In the case of 50% cross-partition transactions, the OCC+2PC system achieves a speedup of 1.07x over Arbor in peak throughput.

Arbor suffers from a relatively higher reduction in throughput as we increase the portion of cross-partition transactions in the workload. The bottleneck in Arbor is that all shards replicate and process cross-shard transactions. This processing stage limits the benefits of different shards processing single-shard transactions in parallel. Figure 5.4(e) shows the latency of processing cross-shard transactions in each batch of transactions. We observe that this latency grows at a constant rate of 6 ms for every increase of 5% of multi-shard transactions and peaks at about 60 ms for the 50% cross-shard workload.

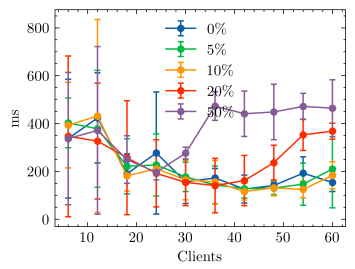
This overhead is intrinsic to Arbor's design because each shard first processes transactions independently, and then all shards aggregate the results to form a single execution



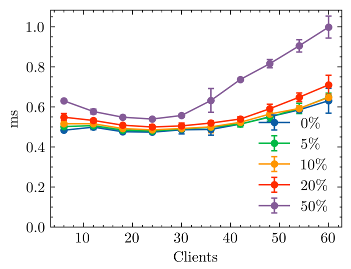
(a) Finalization Throughput



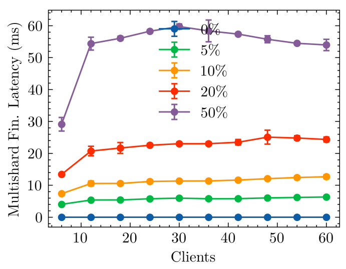
(b) Graph-commit Throughput



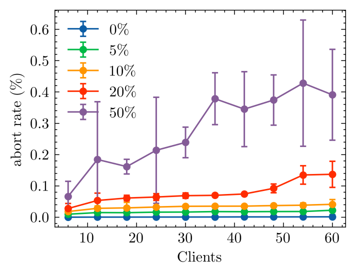
(c) Finalization Latency



(d) Graph-commit Latency

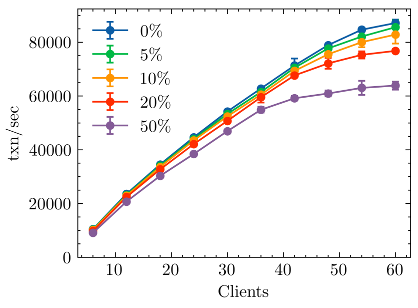


(e) Multi-shard Transaction Processing Latency

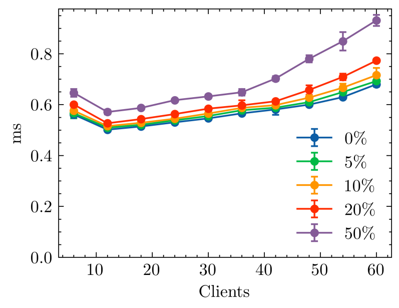


(f) Abort Rate

Figure 5.4: An analysis of cross-partition transactions effect on Arbor using a multi-key RMW YCSB+T workload with uniform key distribution. Three shards are deployed in all experiments. Each graph's legend indicates the percentage of cross-shard transactions in the corresponding workload.



(a) Throughput



(b) Latency

Figure 5.5: An analysis of cross-partition transactions effect on the OCC+2PC system using a multi-key RMW YCSB+T workload with uniform key distribution. Three shards are deployed in all experiments. Each graph's legend indicates how many transactions are single-shard in the corresponding workload by percentage.

branch. Cross-shard transactions cannot be processed the same way because, after aggregation, there could be conflicting updates in a cross-shard transaction's conflict window from transactions processed at a different shard. Consequently, this would lead to a non-serializable execution. As a result, we opted for a design where all shards process cross-shard transactions after collectively determining the order of single-shard transactions.

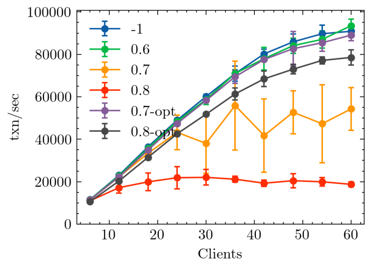
The OCC+2PC system commits cross-shard transaction using 2PC which adds overhead due to the network roundtrip required between the participating shards. The second phase of 2PC is executed lazily in this system. As a result, the transaction coordinator returns the commit/abort decision to the client before the updates of the transaction are made visible. This optimization reduces the overhead of processing cross-shard transactions.

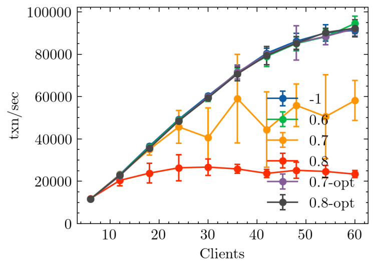
5.5.4 Skew

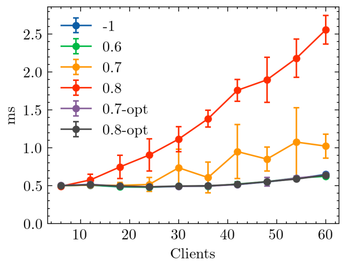
We now analyze the effect of skew on Arbor and the OCC+2PC system. We deploy each system using three shards and use the multi-key RMW YCSB+T workload with a Zipfian key distribution and configure the number of cross-partition transactions in the workload to 5%. We vary the Zipf coefficient of the distribution to alter the level of skew and present our results in Figure 5.6 and Figure 5.7.

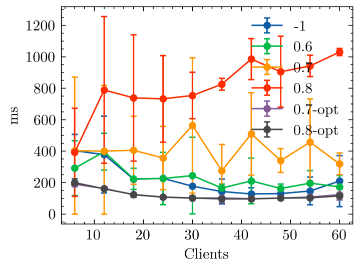
We observe that as the level of contention increases with higher Zipfian coefficient values, the throughput of Arbor decreases. Furthermore, as the Zipfian coefficient increases beyond 0.6, Arbor's throughput decreases as we add more clients after reaching the peak throughput. We found that the primary reason for this drop in performance of Arbor with 0.7 and 0.8 Zipfian coefficients is that the shards receive an unbalanced number of graph-committed transactions. As a result, it takes longer for the shards with a smaller share of requests to receive enough transactions to begin processing a batch of transactions. In turn, the shards receiving a larger share of the requests are blocked during the broadcast and aggregate stage of processing by having to wait for the other shards.

To address this problem, we implemented an optimization where each shard makes an asynchronous RPC to notify the other shards of when it begins to process a batch of transactions. The results are presented in Figure 5.6 with the *opt* suffix added to the respective experiments' legend in the graph. We find that since the 'hot' shards are no longer slowed down by the other shards, the drop in performance is not as significant. A secondary effect of this optimization is that since shards are blocked for lesser time, the overall finalization latency for the optimized experiments is lower as presented in Figure


(a) Finalization Throughput


(b) Graph-commit Throughput


(c) Graph-commit Latency


(d) Finalization Latency

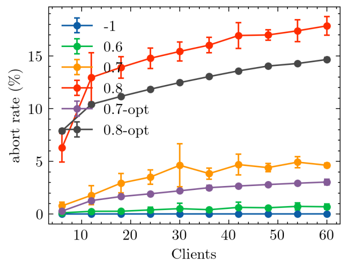
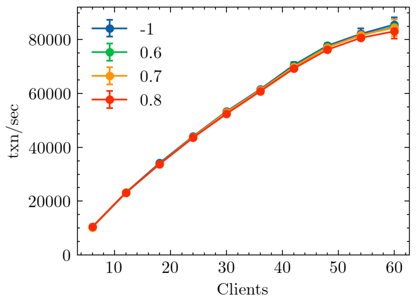
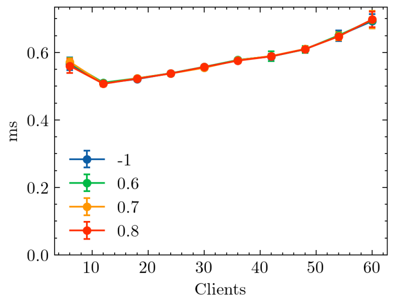

(e) Abort Rate

Figure 5.6: Effect of skewed key distribution on Arbor. The workload consists of multi-key RMW YCSB+T transactions with varying levels of Zipfian skew. The values in each individual figure's legend denotes the corresponding Zipfian coefficient and a value of -1 indicates a uniform distribution.


(a) Throughput


(b) Latency

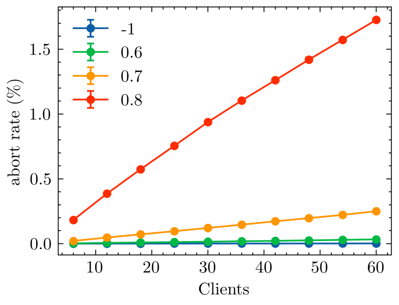

(c) Abort Rate

Figure 5.7: Analyzing the effect of skewed key distribution on the OCC+2PC system using a YCSB+T workload consisting of 5% cross-partition transactions and a multi-key RMW access pattern. We vary the Zipfian coefficient across experiments.

5.6(d). The remaining factor for the lower throughput compared to the uniform distribution case is the higher abort rates as shown in Figure 5.6(e).

Figure 5.7 illustrates the results for the OCC+2PC system for the same set of experiments. We find that the abort rate increases with the level of skew, but to a lesser extent than Arbor. This is because the window of time for conflicts to occur is much smaller. Transactions are processed in the order of a few milliseconds compared to hundreds of milliseconds in Arbor. As a result, there is a higher chance for conflicts to take place. With the optimizations we made, Arbor is able to achieve higher peak throughput than the OCC+2PC system for workloads with up to 0.7 Zipfian coefficient.

5.5.5 Transaction Execution Latency for Applications

Our evaluation has primarily comprised of experiments with fixed workload configurations and a varying number of clients. This evaluation method allows us to saturate each system to determine their respective peak performance. However, invocation load in real-world workloads fluctuate, and as a result, the storage system would not always be saturated. If we consider workflow-like applications that consist of a number of stages, where the output of one stage is used by the next, then the amount of parallelism is limited and the performance is largely determined by the execution latency.

To evaluate the performance on such workloads, we consider a closed-loop setting consisting of 12 clients that sequentially execute transactions using a single-shard deployment of Arbor, VoltDB and the OCC+2PC system. In the case of VoltDB, unlike the experiments shown in Section 5.5.1, clients invoke stored procedures synchronously. If they instead invoked stored procedures asynchronously, they would not have the results of the transaction to use in subsequent computation.

Table 5.1 illustrates the mean transaction execution latency for each system along with the speedup for Arbor relative to the other systems. We observe that Arbor is faster than OCC and VoltDB in all cases, demonstrating the usefulness of its two-round commit model. The read-only latency is higher than the write-only latency for Arbor and the OCC system since each read requires a network round trip whereas writes are buffered on the client until it finishes executing a transaction. Although graph-commits do not provide the same guarantees as synchronous commits, all systems result in a serializable execution of transactions. The primary difference is that with the synchronous commits of the OCC+2PC system and VoltDB, an application client immediately knows whether a transaction committed or aborted. On the other hand, in Arbor, this result is realized

Workload	**Arbor latency**	**VoltDB**		**OCC**	
		Latency	**Arbor speedup**	**Latency**	**Arbor speedup**
Read-only	0.239 ± 0.007	0.540 ± 0.016	2.26	0.333 ± 0.003	1.39
Write-only	0.191 ± 0.0002	0.560 ± 0.009	2.93	0.216 ± 0.004	1.13
1xRMW	0.282 ± 0.003	0.582 ± 0.009	2.06	0.345 ± 0.002	1.22
2xRMW	0.442 ± 0.008	11.536 ± 0.332	26.1	0.531 ± 0.007	1.20
Retwis Load-Heavy	0.156 ± 0.002	7.302 ± 0.839	46.8	0.219 ± 0.0004	1.40
Retwis Post-Heavy	0.216 ± 0.0003	8.896 ± 0.061	41.18	0.284 ± 0.0009	1.31

Table 5.1: Analysis of Arbor's graph-commit latencies, and OCC and VoltDB's commit latencies (in milliseconds) in a closed-loop environment with 12 clients and a single machine server deployment using a range of workloads. For the read-only and write-only configurations, we issue two operations per transaction. We report mean values along with the standard deviation taken from three repetitions of each experiment.

only after a batch of transactions has been processed several milliseconds after the client submitted the transaction and continued with its execution.

Chapter 6

Conclusion

Applications developed on serverless orchestration frameworks typically store global application state on an external storage system. Such storage systems should offer concurrency control since different workflow instances can update overlapping state simultaneously. However, existing concurrency control algorithms can incur significant overhead due to acquiring locks or analyzing read/write sets combined with executing 2PC. This is detrimental to application throughput since storage accesses are on the critical path of client execution.

This thesis presented Arbor, a sharded storage system designed for optimistic execution of serverless workflows while ensuring serializability. Arbor uses a two-round commit model where clients first submit transactions that are added to a dependency graph, and they are then processed in batch off the critical path of client execution. As a result, clients using Arbor can execute at high throughput. Our experiments showed that this commit model can achieve a median speedup in execution latency of 1.26x compared to the synchronous commits of a system that uses OCC+2PC.

To enable this commit model, Arbor uses dependencies included in transactions by clients to track the lineage of updates. It stores data in a dependency graph and reorganizes batches of transactions from it to a validation tree during finalization. Creating branches gives Arbor additional context to decide which transactions to commit relative to approaches that pessimistically abort conflicting transactions. Developers can implement custom policies to select which branch from the validation tree to commit.

Arbor uses a novel approach of partitioning data by key and replicating metadata (i.e., keys and dependencies) of its dependency graph across all partitions. This allows each shard to independently process single-shard transactions while multi-shard transactions are replicated and processed by all shards. We found Arbor's performance to scale by 1.48x and 1.77x when using two and data three shards, respectively. Furthermore, we found that Arbor outperformed an OCC+2PC-based approach in workloads with up to 20% cross-partition transactions.

6.1 Future Research Directions

6.1.1 Developing a Serverless Platform That Leverages Arbor

Our evaluation demonstrated that Arbor has characteristics is scalable and can lower execution latency for applications. These characteristics would be beneficial in serverless workflow applications. A future step that we can take is to develop a serverless platform that uses Arbor as its backend storage. This platform could allow developers to compose applications that execute optimistically and take advantage of Arbor's dependency-tracking interface. Such applications could consist of concurrently running functions that work together that end up with a serializable execution without using expensive coordination or synchronization primitives.

6.1.2 Improving Performance Under High Contention Scenarios

While Arbor's two-round commit model allows clients to continue their execution with low overhead, it suffers when contention is high. This is because transactions are first executed, collected in a batch and then validated asynchronously. In a high contention scenario, this model can suffer from cascading aborts as data is susceptible to becoming stale, leading to a significant number of conflicts. For applications that have relaxed consistency requirements, one solution might be to allow merging of branches during the finalization process.

6.1.3 Durability and Fault Tolerance

Our current prototype of Arbor is an in-memory storage system and does not persist data in secondary storage. A future step would be to persist finalized transactions to provide crash consistency. Furthermore, our prototype does not replicate data items across multiple machines. As a result, if any of the data shards were to fail, then the system would not be able to continue finalizing transactions, nor would it be able to service client requests for data belonging to the failed shard(s). Developing a replication mechanism where each data partition consists of multiple replicas would be useful. To accommodate this design change, only one replica in each partition (perhaps the leader) could participate in the broadcast and aggregate stage of processing each batch of transactions.

References

[1] Amazon Simple Storage Service. https://aws.amazon.com/s3/. [Online; accessed 30-August-2022].

[2] AWS Lambda. https://aws.amazon.com/lambda/. [Online; accessed 20-July-2022].

[3] AWS Lambda Customer Case Studies. https://aws.amazon.com/lambda/resources/customer-case-studies/. [Online; accessed 2-October-2022].

[4] AWS Step Functions. https://aws.amazon.com/step-functions/. [Online; accessed 27-June-2022].

[5] H-Store FAQ. https://hstore.cs.brown.edu/documentation/faq/. [Online; accessed 24-August-2022].

[6] msgp library. https://github.com/tinylib/msgp. [Online; accessed 19-July-2022].

[7] VoltDB Guide to Performance and Customization. https://docs.voltdb.com/PerfGuide/Hello2Async.php. [Online; accessed 27-August-2022].

[8] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: managing datastore locality at scale with akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 445–460, 2018.

[9] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. {CORFU}: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, 2012.

[10] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340, 2013.

[11] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.

[12] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[13] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[16] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1615–1628, 2016.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[18] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. Ycsb+ t: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230. IEEE, 2014.

[19] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment*, 12(2):169–182, 2018.

[20] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, jul 2015.

[21] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment*, 10(5), 2017.

[22] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[23] Richard G Guy, John S Heidemann, Wai-Kei Mak, Thomas W Page Jr, Gerald J Popek, Dieter Rothmeier, et al. Implementation of the ficus replicated file system. In *USENIX Summer*, pages 63–72, 1990.

[24] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[25] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, 2021.

[26] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 152–166, 2021.

[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[28] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[29] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 697–713, 2022.

[30] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[31] Costin Leau. Spring data redis-retwis-j, 2013.

[32] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[33] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: a fast and practical deterministic oltp database. 2020.

[34] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, 2014.

[35] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[36] Robert HB Netzer and Barton P Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[37] Ricardo Padilha and Fernando Pedone. Augustus: Scalable and robust storage for cloud applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 99–112, 2013.

[38] Guna Prasaad, Alvin Cheung, and Dan Suciu. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 527–542, 2020.

[39] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020.

[40] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[41] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[42] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[43] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *Proc. VLDB Endow.*, 11(10):1137–1150, jun 2018.

[44] Chenggang Wu, Jose M Faleiro, Yihan Lin, and Joseph M Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 33(2):344–358, 2019.

[45] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Transactional causal consistency for serverless computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 83–97, 2020.

[46] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*, pages 231–243, 2018.

[47] Linguan Yang, Xinan Yan, and Bernard Wong. Natto: Providing distributed transaction prioritization for high-contention workloads. In *Proceedings of the 2022 International Conference on Management of Data*, pages 715–729, 2022.

[48] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204, 2020.

[49] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[50] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.