NTNU
Norwegian University of
Science and Technology

Durability in a data-flow storage system

Lars Martin Bævre Ek

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Kjetil Nørvåg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Traditional database systems do not meet the throughput demands of today's web applications. Mitigation strategies on the form of intricate cache hierarchies and manual view materialization solve parts of the performance equation, at the cost of increasing complexity. Soup is a new structured storage system that scales to millions of reads per second on a single machine, without the architectural complexity seen in other storage deployments. By propagating updates through a data-flow graph, where pre-computed state is incrementally maintained at materialized nodes throughout the graph, Soup moves the majority of the processing work from reads to writes.

Soup stores all data in volatile main-memory and relies on a write-ahead log for durability. While fast memory access is crucial for frequently accessed materialized views, it is a cumbersome requirement for Soup's base tables, which only serve read requests at rare occasions. By implementing a disk-resident index structure on top of the RocksDB storage engine, this thesis moves Soup from a pure main-memory database to a structured storage system capable of handling datasets larger than its memory size, with only a small decrease in overall write throughput.

With its base tables stored safely on durable storage, Soup can recover from fatal failures by gradually building up its partially materialized views as needed. Similar to a system that recovers with an empty cache, this reduces initial performance while live requests slowly build up Soup's memory-based state. To completely remove the performance degradation of recovery, this thesis also implements a method of performing a global checkpoint of Soup's materialized state. By approaching the data-flow graph as a distributed system, the method performs a coordinated snapshot of all local state, allowing Soup to recover in about a tenth of the time.

Sammendrag

Tradisjonelle databasesystemer møter ikke prestasjonskravene satt av dagens webapplikasjoner. Kompromisser i form av intrikate mellomlagringshierarkier og manuell materialisering av data løser deler av hastighetsproblemet, på bekostning av økt kompleksitet. Soup er et nytt strukturert lagringssystem som skalerer til millioner av lesninger per sekund på én maskin, uten overflødigheter observert i andre lagringsoppsett. Ved å propagere oppdateringer gjennom en dataflytsgraf, hvor forhåndsberegnede resultater opparbeides inkrementelt ved nodene i grafen, beveger Soup mye av prosesseringsarbeidet fra lesing til skriving.

Soup lagrer all data i flyktig hovedminne og skriver oppdateringer til en loggfil for å vedlikeholde data ved fatale feil. Selv om rask minnelesingshastighet er essensielt for mellomlagrede verdier som aksesseres ofte, er det et tungvint krav for kjernetabellene, da disse gjerne sjelden svarer på leseforespørsler. Ved å implementere en diskbasert indeksstruktur over RocksDB-lagringsmotoren, tar denne avhandlingen Soup fra å være et rent minnebasert system, til et strukturert lagringssystem kapabelt til å håndtere mer data enn det har plass til i minne, med kun en minimal nedgang i skrivegjennomstrømning.

Med kjernetabellene lagret trygt på disk, kan Soup gjenopprettes etter feil ved gradvis oppbygging av partielt materialiserte resultater. I likhet med mellomlagringssystemer som starter uten innhold, fører dette til redusert initiell hastighet. Hastigheten økes sakte, men sikkert, ettersom forespørsler etter data fyller opp de minnebaserte mellomlagringslokasjonene. Med mål om å vedlikeholde samme hastighet, implementerer også denne avhandlingen en metode for å gjennomføre et globalt kontrollpunkt av lokal tilstand i Soup. Ved å se på dataflytsgrafen som et distribuert system, tar metoden et koordinert lagringsbilde av data i systemet, slik at Soup kan gjenopprettes på en tiendel av tiden ved feil.

Preface

This thesis was written as the final deliverable for the Master's of Science in Computer Science program at the Norwegian University of Science and Technology. The program includes a master's thesis—presented here—and a preliminary project, where the former builds upon research of the latter, which was completed in December 2017. Both the preliminary project and this thesis contributes to the Soup research project at MIT's Parallel and Distributed Operating Systems Group and is advised by Kjetil Nørvåg at NTNU's Department of Computer Science.

Acknowledgments

Neither this thesis, nor the research performed in the preliminary project, would have seen the light of day without the steadfast support, advice, and feedback from Jon Gjengset, Malte Schwarzkopf, and the rest of the Soup team at MIT's Parallel and Distributed Operating Systems Group. Thank you for proposing interesting challenges, providing servers for benchmarks, following up on both technical and academic questions, and most importantly, for letting me contribute to Soup—it has been a privilege.

Finally, I would like to thank Kjetil Nørvåg for both excellent advice and guidance throughout the last year.

Contents

Chapter 1

Introduction

Building sophisticated web applications while scaling to potentially millions of users forces developers to compromise between performance, user requirements, and application complexity. Whereas traditional relational databases logically are able to fulfill the increasingly complex storage demands of today's internet businesses, they are far from able to do so at the scale and performance required. To continue serving requests at increasing throughput targets with low latency, developers introduce mitigation strategies ranging from complex cache hierarchies [60] to denormalized schemas [68].

These methods are usually used to drastically improve read performance, while penalizing write throughput and increasing application complexity. Soup [38] sets out to solve this dilemma once and for all, with a structured storage system capable of horizontally scaling to millions of reads per second, without the need for complex cache deployments or manual maintenance of materialized views.

Soup achieves this through use of an incrementally maintained data-flow graph. New updates propagate through the graph at write-time, with pre-computed results stored at selected *materialized* nodes throughout the graph. This moves the bulk of the workload from reads to writes, by giving read operations direct access to computed state from materialized nodes further down the graph.

Soup ensures durability by persisting all updates to a write-ahead log before they are injected into the data-flow graph. While appending entries to a file is good for performance, recovering from an ever-growing log after a failure is far from feasible. This thesis improves Soup's durability situation with two main contributions: it moves Soup's otherwise in-memory table structures to durable storage and implements snapshotting of Soup's materialized views. Both contributions were implemented in the open-source Soup prototype written in the Rust programming language, and a list of the changes is available in appendix A.

1.1 From main-memory to durable storage

After updates are persisted to Soup's write-ahead log, they are injected into the first nodes in the data-flow graph: the base tables. Unlike the partially materialized nodes further down the data-flow graph, the base tables can never be evicted from, and must together always contain a full representation of a Soup application's state. On the other hand, the base tables should only be responsible for serving a small part of Soup's read queries. The rest should be handled by materialized nodes towards the bottom of the graph, using state that was pre-computed when the updates propagated through the data-flow graph.

This makes volatile main-memory a poor destination for Soup's base table data. Applications where data is continuously inserted would cause Soup's memory footprint to grow continuously over time, until eventually reaching its host system's memory limit. Moving the base tables to durable storage avoids this problem, while reducing Soup's overall memory usage.

Storing base tables on durable storage also improves Soup's recovery situation, by avoiding the need to replay the entire write-ahead log after failures. With all updates safely persisted to and readily available from durable storage, recovery is instead a matter of replaying data from the base tables when needed.

1.2 Snapshotting materialized views

With durable base tables, Soup recovers significantly faster than by having to replay the entire write-ahead log. This is not without downside however: whereas log-based recovery brings all nodes in the data-flow graph back to a pre-failure state, durable base tables leave partial nodes empty, resulting in a latency penalty for initial read-queries. Instead, we would like to periodically write *snapshots* of the materialized state at each node to durable storage, ensuring a speedy recovery process for both base tables and materialized views alike.

To snapshot nodes individually while maintaining consistency, it is crucial that all nodes snapshot the same window of updates. For a given update at any given time, said update must either be contained in every snapshot across the graph, or neither of them. While updates are processed synchronously within a single domain in Soup (a partition of nodes), data flows asynchronously between domains, where the boundaries can be both within a local machine and across a network. At the same time, taking a snapshot should not incur a significant pause in processing, which would result in lower throughput all around.

By approaching the problem from the viewpoint of snapshotting in a distributed system, this thesis implements a snapshotting method capable of creating a logically consistent snapshot across the data-flow graph, with a focus on maintaining as much of Soup's performance guarantees as possible.

1.3 Outline

The rest of this thesis is structured as follows:

- **Chapter 2** introduces core theory behind fundamental concepts used in the rest of this thesis.
- **Chapter 3** reviews ideas from research and industry relevant to the thesis' main contributions.
- **Chapter 4** describes new and existing benchmarks used throughout the thesis.
- **Chapter 5** outlines the requirements for a persistent base table implementation, followed by two implementation iterations.
- **Chapter 6** gradually builds up a snapshotting implementation.
- **Chapter 7** evaluates the resulting implementations from the two previous chapters, using the benchmarks introduced in chapter 4.
- **Chapter 8** presents possible next steps towards a production-ready Soup while concluding on the results presented earlier in the thesis.

Chapter 2

Background

This chapter describes various concepts relevant to the rest of this thesis, starting with an introduction to the system the contributions are implemented in, Soup. Afterwards, and outline of the Rust programming language follows, together with a look at the `bincode` serialization library and a few other core technical concepts used throughout the thesis. Finally, an overall view of the two different storage engines SQLite and RocksDB, used to implement the durable index structure described in chapter 5, is given.

2.1 Soup

Soup [38] is an on-going research project at the Parallel and Distributed Operating Systems[1] group at MIT CSAIL. The current Soup prototype is written in the Rust programming language and made available as open-source code on GitHub[2].

This section introduces Soup's core concepts.

2.1.1 Data-flow

Applications using Soup define base table schemas and a set of queries ahead of time. Whereas the former is common in traditional relational database management systems, the latter is not, and is the primary source of Soup's read performance improvements. A relational database computes the result of queries on-the-fly, by building a query-graph, which it then executes. This requires potentially costly computations to be performed multiple times for separate queries, while throwing away intermediary results that could be re-used. Soup instead builds a *data-flow* graph from its pre-defined queries, propagating each update through it at write-time. Computations can then be incrementally maintained on each write, reducing the work needed by a read-operation to something more similar to a simple key-value read in a caching system.

```
CREATE TABLE Car (id int, brand varchar(255), PRIMARY KEY(id));
QUERY CountCars: SELECT COUNT(*) FROM Car WHERE brand = ?;
```

Listing 2.1: An example base table with a corresponding query.

Implemented naively, incrementally maintaining a data-flow graph for each query would have disastrous storage consequences. Each query would need a separate graph, duplicating data across a range of nodes. Instead, Soup builds a single data-flow graph from all of its queries, recognizing common sub-expressions where possible [34]. That still leaves the issue of what state to incrementally maintain. With queries consisting of a potentially large amount of nodes, materializing data at each step would lead to significant overlap between nodes. Instead, Soup only materializes and incrementally maintains state at nodes it considers *stateful*, with other nodes referring upwards in the graph to its closest materialized ancestor node.

[1] https://pdos.csail.mit.edu/
[2] https://github.com/mit-pdos/distributary/

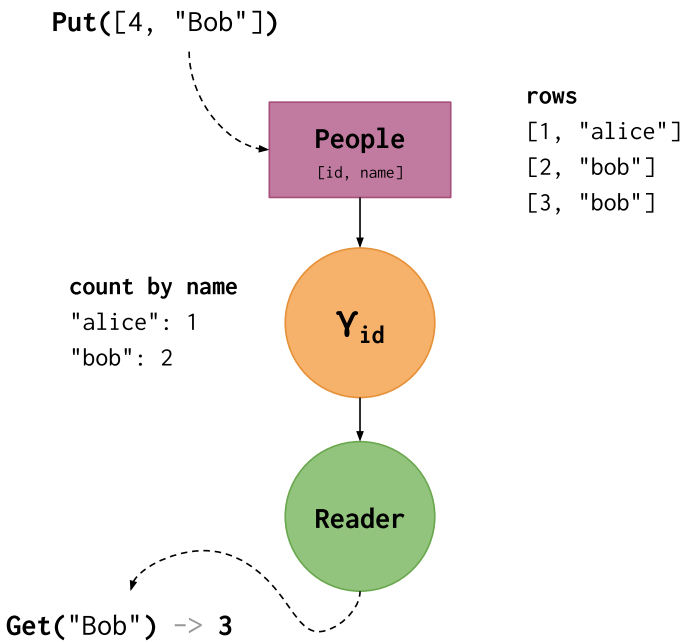

Figure 2.1: A simplified version of a Soup data-flow graph for a program that counts the number of people with the same name. Insertions propagate from top to bottom, where data is materialized at the necessary points. While all rows are stored in the `People` base node, the aggregation operator (γ) only retains the total count for each name, enabling efficient reads.

The state at these materialized nodes is also kept *partial* whenever possible. Instead of storing the results for all queries—like a materialized view does—Soup only retains state for records in the application's working set, evicting rarely used data. Queries for missing keys result in ancestor queries (*replays*) to nodes further up the graph. Similar to cache misses in other systems these eventually propagate all the way up to the base nodes, where a full copy of the state is always maintained.

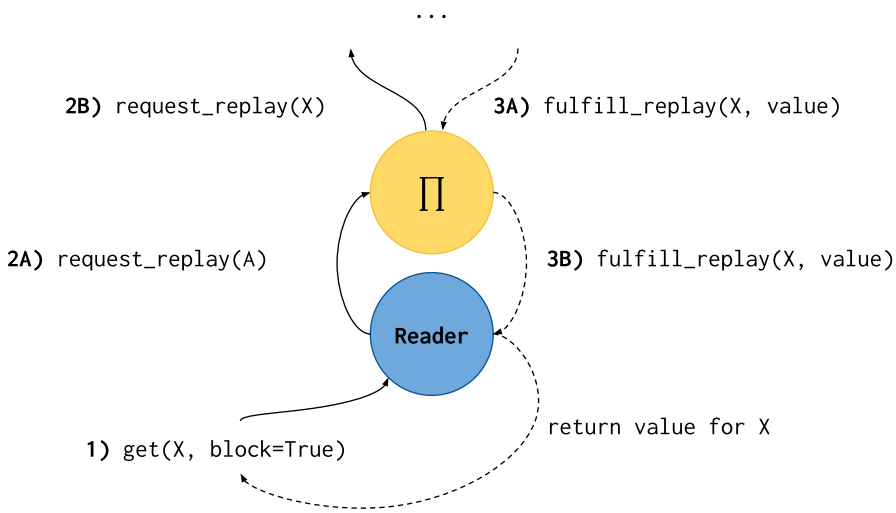

Figure 2.2: Queries for missing keys, in this case `X`, are fulfilled by requesting replays from ancestors in the data-flow graph. Subsequent requests for `X` will be fulfilled by the reader directly, avoiding the need for further replaying.

Migrations

Schema migrations are inevitable in long-running applications: business requirements change, projects are refactored, and new features are added. Performing migrations in traditional database systems, without downtime, is on the other hand far from trivial [36, 80]. While Soup requires both the schema and all queries to be defined ahead of time, it handles changes in both seamlessly.

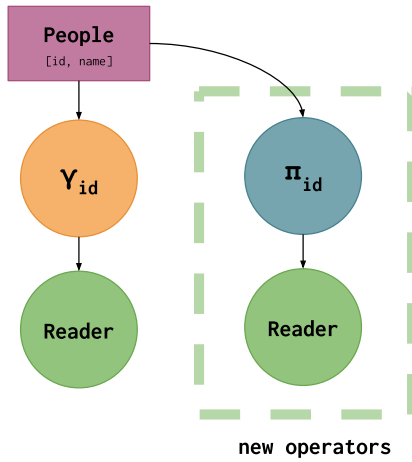

Figure 2.3: To extend the program from figure 2.1 with a query selecting rows by ID, Soup adds a projection operator and a reader to the existing data-flow.

Added queries extend the existing data-flow with additional nodes, while reusing as much as possible of the existing graph. New partially materialized nodes can start serving requests right away, by fetching data from ancestor nodes if necessary. This is not the case with fully materialized nodes, which require a full representation of their state at all times. To bring these nodes online, Soup incurs a full replay of the state needed, potentially delaying new requests until all replays have completed. Review of the migrations performed during the lifetime of the HotCRP conference review program [48]—which uses MySQL—showed that these delays were rare, with Soup being able to transition 95% of schema changes without downtime.

Changes to the base table schema happen in-place, and Soup's base nodes retain a full history of columns for each table. This ensures that both existing and new requests can be served alongside each other, allowing the data-flow graph to transition without downtime.

Domains

Soup's data-flow graph is partitioned into *domains*, each containing a series of nodes. Updates are processed at separate domains asynchronously, in different computational units—threads, or other machines altogether for distributed Soup. Within a single domain, packets are processed synchronously, one at a time, removing the need for locking within domains. Packet processing at domains include both regular updates and other types of packets, such as replay requests.

Domains are separated by egress and ingress nodes, responsible for maintaining communication across domain boundaries using a buffered channel. When Soup is run

in a distributed fashion, communication between separate domains happen over TCP sockets, between the egress and ingress nodes.

Sharding

Distributing domains across computational units lets Soup divide its processing load between a cluster of machines. This is only an improvement if the load is uniformly spread across all domains. When that is not the case, and a majority of the data is skewed towards a single domain, Soup is left to processing most of the requests in a single computational unit. This is where sharding comes in. By splitting the atom that is a single domain into multiple shards, both the computational load and the data stored at that domain can be spread between multiple computational units.

Balancing the data across a cluster is essential in scaling Soup to larger datasets. Without sharding, Soup's capabilities would be capped at the memory size of the largest machine in the cluster. Soup shards data by hash-partitioning keys statically. This is unfortunate for workloads skewed towards a small key subset, where only a few domains might end up serving most of the requests received. Dynamic sharding would evenly re-balance the workload across the shards, and is a future implementation goal for Soup.

When necessary, a *sharder* node is inserted between domains, translating between sharding schemes in separate domains. This allows nodes further down the graph to remain partial, at the cost of having to replay state across domains.

Eviction

To ensure that partial state does in fact remain partial over time, Soup evicts data when necessary. Currently this happens when Soup's memory-limit goes beyond an application-defined limit. This triggers an eviction notice sent to the largest domain—measured in state-size—which then takes care of propagating this eviction notice downstream in the graph to any nodes that might depend on the evicted state.

The keys to evict at a specific node are picked randomly. In the future, this could be improved through more sophisticated eviction strategies, such as only evicting the least recently used records.

2.1.2 Operators

Soup's data-flow graph consists of relational operators, where each operator emits either the `Positive` or the `Negative` records required for downstream nodes to maintain their state accordingly.

Base nodes

Every packet that reaches Soup is first injected into an appropriate base node, similar to a table in a relational database. This is where external API requests are translated into a language the rest of the data-flow graph understands. While clients might issue *e.g.*, deletion requests by *key*, the base nodes translate the request into a `Negative` record for the entire row, which the rest of the data-flow graph can use to invalidate removed state. Update requests are likewise first translated into a `Negative` record, followed by a `Positive` record containing the new row.

Whereas other stateful nodes can keep their state *partial*, choosing which records to maintain and which to discard, base nodes always remain fully materialized at all times.

Stateless operators

Stateless operators process updates with no regard for prior events, without maintaining any state at all. The operators that fall within this category are *pure functions*, such as the *projection* and *filter* operators. The former picks out one or more fields from each incoming row, while the latter determines if a row should be forwarded through the data-flow graph or not.

While Soup is free to insert operators into its data-flow graph as it sees fit, both projections and filters can often be mapped directly to a part of an SQL query. `SELECT name ...` would result in the projection shown in figure 2.4, while a `WHERE`-clause on the form of `WHERE age < 10` would produce a filter operator emitting only records where `age < 10` is true.

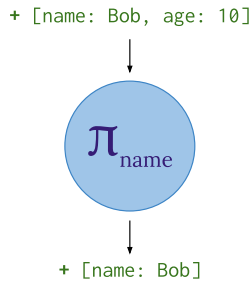

Figure 2.4: A projection operator responsible for picking out the `name` column of incoming records.

Stateful operators

Whereas stateless operators resemble their counterparts in relational databases [17], Soup's stateful operators compute results incrementally by maintaining internal state

between updates. This saves Soup from re-doing potentially expensive computations, by instead mutating the previous result when a new record arrives. The *count* operator—produced by a SQL `COUNT`-clause—is an example of a stateful operator, where `Positive` and `Negative` records incur an addition or subtraction of the current count, respectively. Another example is the *top-k* operator, produced by *e.g.*, an `ORDER BY`-clause coupled with a `LIMIT` to determine the k most significant or insignificant elements.

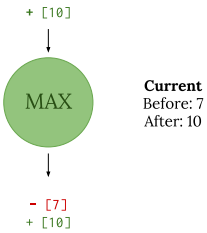

Figure 2.5: The *max* operator emits both a negative and a positive record when its state changes: the former to signal that its previous state should be invalidated, the latter to inform downstream nodes of the new result.

Joins

Finally, Soup supports joining together multiple paths of the data-flow graph. This is made possible using *ancestor queries*. Whenever a record arrives at a join operator, it queries its other ancestors for the records required to produce a single, unified update. To make sure that this is completed in an efficient manner, join operators force their ancestors to retain indices on the fields that they need to be queried for.

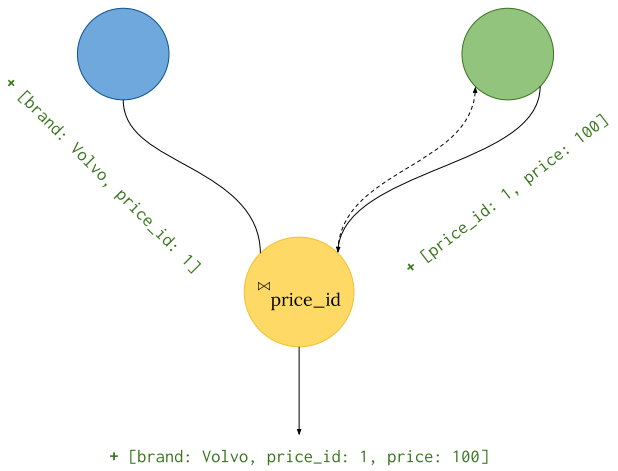

Figure 2.6: An ancestor query is performed to the right side of the join to produce a unified record for the update received from the left.

2.1.3 Eventual consistency

Databases are often considered the source of truth for applications, and anomalies here could have disastrous consequences. Whereas fatal failures are easy to recognize, unexpected behavior at the data layer could be the exact opposite. This is a convincing argument for strong consistency, where the result of all operations can be reasoned about from the ordering and type of operation performed.

Regardless, companies scaling web applications to large amounts of users often opt for systems with lesser consistency guarantees [18, 20, 79]. While the performance of strongly consistent systems have taken a turn for the better with the introduction of horizontally scalable systems with clearly defined consistency guarantees [14, 19], the race is still far from even when compared to systems with lesser consistency guarantees. At the same time, the question of whether strong consistency is actually a requirement for most applications remain. Analysis of live requests at Facebook [53] showed the opposite: only 0.0004% of reads would have returned different results in a strongly consistent system with total ordering. These cases could then be handled explicitly, avoiding the need to penalize the performance of an entire system for a fraction of the requests.

Soup targets applications where eventual consistency is sufficient, and would not be able to provide the performance it does without it. Eventual consistency avoids the need for explicit synchronization on every update, while allowing Soup to scale in a distributed fashion without a total ordering of writes. Clients receive write acknowledgments for writes when updates have been safely persisted to durable storage, after which they are propagated through Soup's data-flow graph asynchronously. Reads access

double-buffered hash tables directly [37], without the need for locks. Writes update one of the buffers and expose updated content to the readers with an atomic swap.

2.1.4 Architecture

Soup is composed of a series of components and runs distributed on an arbitrary number of machines. Each Soup cluster elects a single controller, which serves as an entry point for further communication. The bulk of the processing work happens within workers and readers.

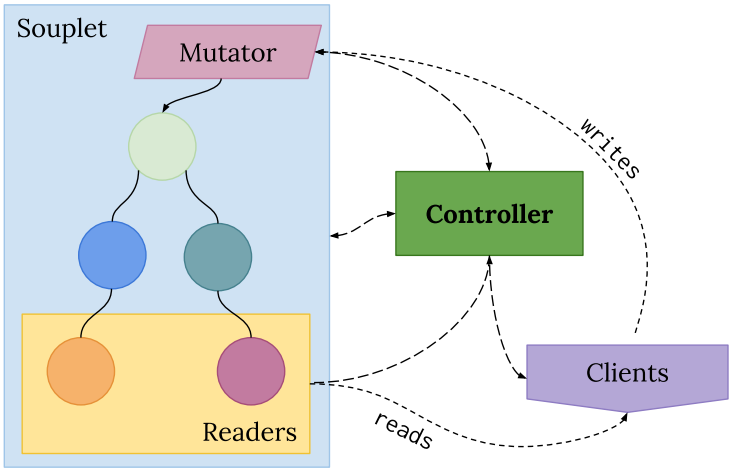

Figure 2.7: Clients communicate directly with readers and mutators for reads and writes.

Controller

At the heart of Soup lies a replicated controller, with a leader elected using ZooKeeper [42]. The controller is the first point of contact for Soup's external APIs, and is responsible for managing Soup's data-flow graph. In the face of a migration, the controller issues commands to the workers, to modify the graph as necessary to continue serving requests.

Souplets

Processing of updates happen within the Souplets—Soup's worker nodes. Each Souplet includes a pool of threads which together go through the incoming packets for its domains. While maximum one thread can process updates for a domain at once, multiple domains can process updates in parallel. Earlier versions of Soup ran domains in separate

threads altogether, resulting in a core-constrained system when the number of domains went far beyond the host's CPU core count.

Communication between the Souplets and the controller happens over TCP, at the coordination layer.

Mutators

After contacting the controller to construct a mutator, clients inject updates directly into the Soup data-flow graph, without further controller communication. Each base node requires a separate mutator. The mutator API includes methods for inserting, deleting, and updating records. Clients receive acknowledgments when writes have been successfully persisted to Soup's write-ahead log and durability can be ensured.

Readers

Soup's reader nodes make use of double-buffered hash tables to make it possible to read and write to a single data structure at the same time, without locks. To prevent high throughput write processing from slowing down reads, read requests are processed by separate threads—readers. Whenever a reader lacks state for a specific key, it issues a replay request to its ancestor nodes, which will be served on Soup's regular update path. Finally, the completed replay updates a reader's state by swapping the double-buffered tables.

2.1.5 Interacting with Soup

Applications using Soup define a base table schema and a set of corresponding queries. The query syntax resembles that of prepared statements in relational databases, where placeholders are replaced with values when the query is used in a read operation. Both the schema and the queries can be modified and extended later on, through Soup's external API.

```
/* Base table schemas: */
CREATE TABLE Article (aid int, title varchar(255),
                      url text, PRIMARY KEY(aid));
CREATE TABLE Vote (aid int, uid int);

/* Intermediate view (not exposed through the client): */
VoteCount: SELECT Vote.aid, COUNT(uid) AS votes
            FROM Vote GROUP BY Vote.aid;

/* Read query: */
QUERY ArticleWithVoteCount:
            SELECT Article.aid, title, url, VoteCount.votes AS votes
            FROM Article, VoteCount
            WHERE Article.aid = VoteCount.aid AND Article.aid = ?;
```

Listing 2.2: Soup schema with two base tables and an external query.

Writing to and reading from Soup is done through mutators and getters. Both are built by going through the controller, after which writes can go directly to the domain and reads can access readers directly.

```
// Build mutators and getter.
let mut article = blender.get_mutator("Article").unwrap();
let mut vote = blender.get_mutator("Vote").unwrap();
let mut awvc = blender.get_getter("ArticleWithVoteCount").unwrap();

// Insert a new article:
let aid = 1;
let title = "new article";
let url = "https://ntnu.edu";
article
  .put(vec![aid.into(), title.into(), url.into()])
  .unwrap();

// Vote for the article:
let uid = 123;
vote
  .put(vec![aid.into(), uid.into()])
  .unwrap();

// Read the vote count:
println!("{}", awvc.lookup(&[aid.into()], true));
```

Listing 2.3: Soup example usage, where an article and a vote is inserted, followed by a read of the vote count.

Data-flow graph

As an example, let us consider the data-flow graph Soup generates for the program in listing 2.2, shown in figure 2.8. Each box in the graph denotes a node, and a separate color is used for each domain. Updates are injected into a base node—either `Article` or `Vote`—before they propagate through the graph. Both base nodes are fully materialized and serve as the source of truth for all data inserted into Soup. Continuing, we arrive at the egress and ingress nodes shown on the second level. These act as connectors between separate domains, and facilitate communication either within a single Souplet or across machines if Soup is distributed.

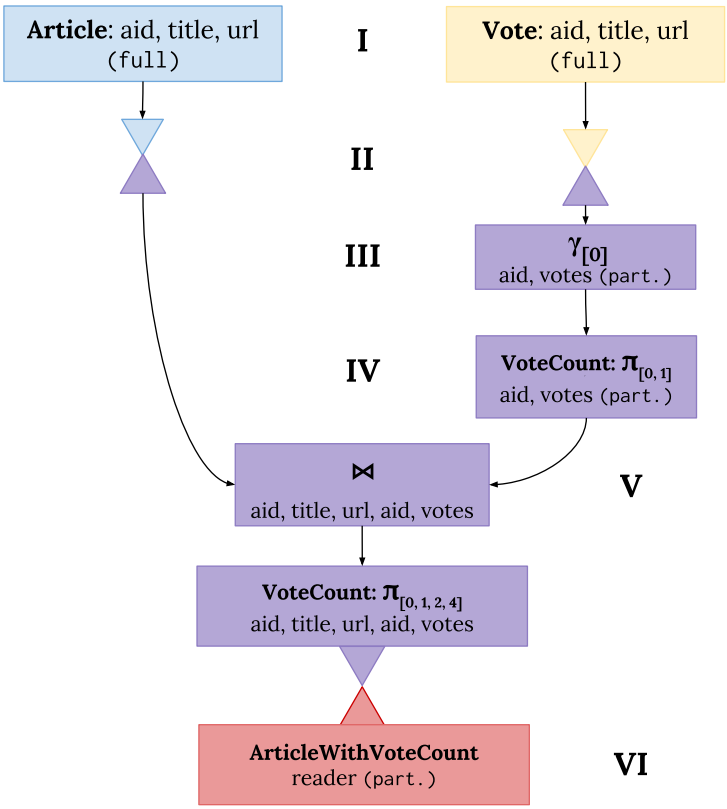

Figure 2.8: The data-flow graph produced by Soup for the example program shown in listing 2.2. Nodes are marked with either `(full)` or `(part.)`, depending on whether their state is fully or partially materialized.

The third label shows the first operator in the data-flow graph, an aggregator. Described by the gamma symbol (γ), it performs the work of our `COUNT(uid)` clause. The aggregation is incrementally maintained, and the rest of the graph is notified of changes in the total count when the operator first emits a negative record—stating that the previous

count is no longer valid—followed by a positive record for the updated value. The fourth level contains a projection operator, responsible for picking out the correct fields in the correct order from its parent node.

Moving along, we reach the join operator responsible for combining rows from the `Article` and the `Vote` tables. Notice that the node is neither partially nor fully materialized—it contains no state at all. Whenever an update arrives from one of its parents, it requests the rows necessary to perform a join from the other. Similar to the aggregation, the join node is also followed by a projection, which in this case simply emits all the fields from the join.

Finally, we reach the reader node at the bottom of the graph labeled with the external query it serves requests for: `ArticleWithVoteCount`. While updates begin at the base nodes at the top, reads begin at the bottom. The reader here is partially materialized, letting it serve subsequent requests for the same key without further ado. Reads for keys missing in its state result in replays, which will propagate upwards through the graph until a node with materialized state for the specific key is reached. The replay then flows downwards again, until it reaches the reader, where the result is returned to the client.

2.1.6 MySQL Protocol Translation

Soup supports a decent subset of SQL in its query definitions. Regardless, using Soup in an application requires significant changes: all queries have to be defined ahead of time, and interactions with Soup have to go through Soup's external API. Soup's MySQL shim [64] makes this easier by letting applications communicate with Soup using the MySQL binary protocol.

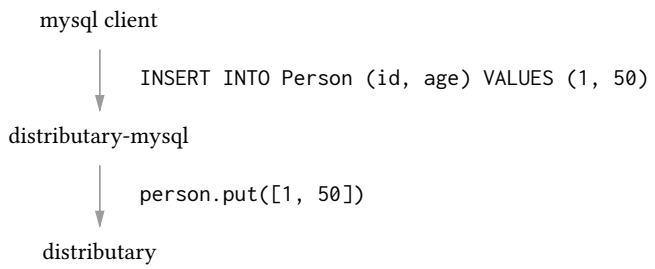

Figure 2.9: `distributary-mysql` translates SQL-queries to appropriate Soup API-calls.

The MySQL shim acts as a separate service, which clients interact with over TCP. Received queries extend the Soup data-flow graph if needed, before they are forwarded to an appropriate Soup worker for execution.

2.2 SQLite

SQLite [72] is by far the most widely deployed database ever written. Used in everything from smart phones to cars, with an estimated user count in the magnitude of multiple billion users, SQLite is everywhere[3]. SQlite is an embedded database, and requires no extra processes, or even threads, to run.

In a world of unreliable software, SQLite is stable as a rock. It has 100% branch test coverage, with a test suite containing millions of different test cases. SQLite is, and always has been, available in the public domain. As the name implies, SQLite provides an SQL interface to developers, with decent support for everything from indices to views. The library itself is written in about 130 thousand lines of C code.

While the main usage of SQLite is as a persistent application store (*e.g.*, in browsers and mobile applications), SQLite is also popularly used as an engine in other databases. One such example is the recently open-sourced FoundationDB [4], which provides a distributed database with full ACID transactions, where each shard makes use of SQLite at its core.

2.2.1 B-trees

Similar to a significant amount of other relational databases, SQLite makes use of B-trees [7] for its on-disk index structures. This is with good reason: B-trees are well suited for mediums that perform better with larger blocks of data, such as traditional spinning hard drives. While it has never been officially decided what the B in B-tree stands for, a B-tree is a self-balancing binary tree data structure.

Unlike other tree structures, such as binary search trees, each node in a B-tree holds multiple values. By keeping the amount of values in a node—the node size—close to the size of a block on disk, most of a B-tree's operations can be performed in $O(\log_b n)$ disk reads, where b is the maximum number of entries per block, and $\log_b n$ the height of the tree. With traditional storage mediums, where a single disk seek might take multiple milliseconds, this is extremely important.

When the term B-tree is used in database systems today, it is usually used to refer to an improved version of the traditional data structure, and known as a B+tree. Whereas the former stores values in all levels of the tree, the specialized version only does so at the leaf level, with the internal nodes only containing copies of the keys. Actual records can then be stored in a different on-disk data structure, with pointers from the leaf nodes, and by introducing sibling pointers at the leaf node level, range queries can be efficiently executed by walking the bottom of the tree horizontally.

[3] Who uses SQLite? `https://www.sqlite.org/mostdeployed.html`

2.2.2 Rollback journal

SQLite implements support for atomic transactions through the use of a rollback journal. A historic copy of values prior to changes are kept in a separate file—the rollback journal—so that they can be copied back to the actual database file in the event of a `ROLLBACK`. Similarly, this file can be deleted after a `COMMIT` of the transaction.

With a rollback journal, SQLite requires a full exclusive lock to be held for the duration of all mutations to prevent file corruption, blocking any potential readers from accessing the database. This is the main reason SQLite is commonly not used as the storage system for applications that require high-performance concurrent access to their database (*e.g.*, web application backends with multiple active users): only a single write operation could be performed at the time. This is not the case with reads, which hold shared locks.

2.2.3 Write-ahead log

In version 3.7.0, SQLite introduced an alternative to the traditional rollback journal: the write-ahead log [73]. Maintaining the same atomicity and durability guarantees, the use of a WAL significantly improves write performance by catering to more sequential disk access. Additionally, reading can now co-exist with writing, as writers do no longer block read access.

The original rollback journal format writes directly to the database file, while maintaining old values in the rollback journal. SQLite in WAL-mode does the opposite. Updates are appended to the WAL, and copied over to the main database file when a *checkpoint* is taken. This is also the reason readers can continue to access the database while writes are happening, as the database file itself is not mutated, only the WAL.

This introduces a slight performance penalty for reads, however, as there are now potentially two sources of truth for all content: the main database file, and the WAL—until a checkpoint happens. The longer the WAL is, the more time has to be spent searching through it by reads.

2.2.4 Interacting with SQLite

Most applications interact with SQLite through its C-API, compiling SQL queries into prepared binary statements, which can then be executed efficiently with different arguments, as shown in listing 2.4.

```
sqlite3 *db;
sqlite3_stmt *statement;
char *err_msg = 0;
sqlite3_open("test.db", &db);
sqlite3_exec(
  db,
  "CREATE TABLE data (id INTEGER PRIMARY KEY)",
  NULL,
  NULL,
  &err_msg
);

// Compile a prepared statement:
sqlite3_prepare_v2(
  db,
  "INSERT INTO data VALUES (?1)",
  -1,
  &statement,
  0
);

// Then insert a single row with the value 10:
int id = 10;
sqlite3_bind_int(statement, 1, id);
```

Listing 2.4: Simple SQLite C-example showing how to write a single row (error handling ignored for brevity)

In addition, most programming languages have at least one popular library for accessing SQLite, abstracting away the need to directly call into the C-bindings through more idiomatic APIs for each language. SQLite also provides a command-line interface, which can be used for reads and modifications.

2.2.5 SQLite from Rust

Accessing SQLite from Rust can be done through the excellent `rusqlite` library [35], which provides a Rust API on top of SQLite's C-bindings.

```
let conn = Connection::open("test.db").unwrap();
conn.execute(
  "CREATE TABLE data (id INTEGER PRIMARY KEY)",
  &[],
).unwrap();

// Compile a prepared statement:
let statement = conn.prepare("INSERT INTO data VALUES (?1)").unwrap();

// Then insert a single row with the value 10:
let id = 10;
statement.execute(&[&id]).unwrap();
```

Listing 2.5: SQLite example using `rusqlite` showing how to write a single row.

2.3 RocksDB

RocksDB is an embedded key-value store optimized for modern flash storage. RocksDB started out at Facebook, with the goal of making a version of Google's LevelDB that performed well on modern hardware. Today, RocksDB is used at the heart of a wide variety of databases, such as CockroachDB [14], MyRocks [31] and TiDB [65].

Traditional B-tree based database systems are often faced with poor write performance as a result of random writes, which perform worse than sequential writes on both magnetic and flash-based storage mediums. RocksDB, on the other hand, achieves impressive write performance through the use of immutable log-structured merge trees [61] (LSM-trees), avoiding the need for random writes to persistent storage altogether.

Writes are initially only written to a persistent write-ahead log (WAL) and in-memory data structures referred to as memtables. Later, these memtables are flushed to their equivalent data structures on disk, Static Sorted Tables (SST). The latter is done by background threads, allowing regular processing to continue without getting blocked by slow writes to persistent storage. Both of these components originate in Patrick O'Neil's original paper on LSM-trees [61], where the in-memory data structure is referred to as C_0, and the on-disk structures $C_{1..n}$.

2.3.1 MemTables

All writes are initially synchronously written to an in-memory data structure—a memtable—which is later flushed to disk at the point of filling up. Both the size and the number of memtables can be configured at runtime.

RocksDB's default memtable implementation is a skiplist, with an $O(\log n)$ bound on inserts, searches, and deletes. This can be changed to a series of hash based implementa-

tions, which offer better performance if all operations are done within a pre-specified key prefix.

2.3.2 Static sorted tables

After a memtable reaches a certain size, RocksDB's background threads takes over and flushes it to persistent storage. This will generate one or more SS-tables on disk, where each file is sorted. SS-tables are immutable: a new SS-table is always created, and existing ones are never updated. This ensures that writes remain sequential.

2.3.3 Write-ahead log

RocksDB achieves durability through the use of a write-ahead log (WAL). Without it, data in memtables would be lost at the event of a crash. By default, every `Put` operation results in a write to the RocksDB WAL, with the optional possibility of waiting for the write to be fully synchronized to the WAL before returning.

Each memtable corresponds to a WAL-file, which is marked as obsolete when the memtable has been safely persisted to disk. Each WAL-file includes a sequence number, and the files are iterated through in order during recovery. The WAL itself is built up of a sequence of records, where each record includes a computed cyclic redundancy check hash over the payload, to maintain integrity [33].

Optionally, the WAL can be written to a different disk than the regular database files. This is essential for production systems that want to maintain a high write throughput: compactions and memtable flushes can then utilize the full disk capacity without slowing down the throughput of WAL writes. Even more drastically, the database files could be written to faster, volatile storage, relying solely on never-archived WAL-files for (albeit much slower) recovery.

```
let batch = WriteBatch::default()
batch.put("a", "1");
batch.put("b", "2");

let opts = WriteOptions::default();
opts.set_sync(true);
db.write(batch, &opts);
```

Listing 2.6: Rust code for safely persisting a batch of writes to RocksDB and its write-ahead log.

`Put` operations can also be batched into a `WriteBatch` (as shown in listing 2.6), to amortize the cost of synchronizing the WAL over a larger amount of write operations. This is an atomic operation: either all the writes in the write batch succeed, or none do.

2.3.4 Basic operations

Akin to other key-value databases, RocksDB offers a familiar API of `Put(key, value)`, `Get(key)` and `Delete(key)`, operating directly on bytestream values. Both insertions and deletions are purely sequential: subsequent `Put` operations of the same key never backtrack and overwrite existing keys, and deletions insert tombstone markers to avoid having to randomly read and mutate previously written values.

Whereas both memtables and SS-tables are sorted, each tree structure has the possibility of overlapping with another. This is a result of the immutability property, and newly created SS-tables might contain key ranges already included in existing structures. This means that read operations in RocksDB, and other LSM-tree based storage systems, have to iterate through each tree structure—starting with the memtables—in an attempt to find the key in question. Reads within each sorted tree structure can be done in $O(\log n)$ through a binary search. Going through a potentially large amount of SS-tables on disk is costly however, and RocksDB employs a series of tricks to avoid doing so.

2.3.5 Compactions

To maintain immutability, new SS-tables are always created without modifying existing on-disk content. Two writes to the same key can thus co-exist in different SS-tables, even if only the last written key is relevant to the system. This is quite wasteful, and would lead to worse and worse read performance over time. The original LSM-paper [61] solves this through *merging* existing LSM-trees into new ones at regular intervals. RocksDB does so in background threads, where it is referred to as *compaction*.

During compacting, multiple SS-tables are merge-sorted into a single new structure. This process also removes duplicate keys, retaining only the last value for future use. Tombstones are also filtered out, together with any values they might have deleted.

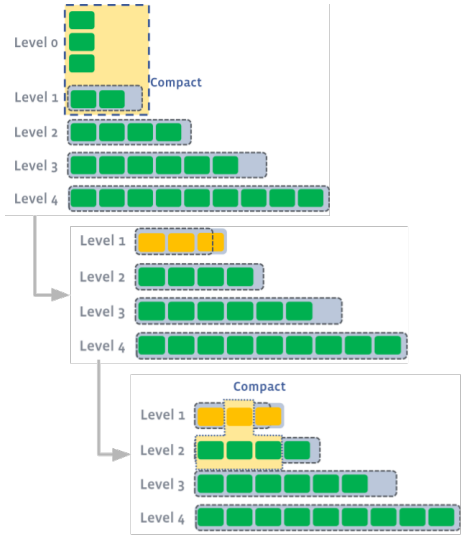

Figure 2.10: SS-tables from initial levels are compacted into the next [30].

In RocksDB, compactions are triggered when the previous *level* reaches a certain size. Referred to as *leveled compaction*, this was one of the original contributions of LevelDB. As described in [30], compactions are usually initiated when the SS-table count at the first level, level 0, goes beyond a certain amount. This in turn might cause the next level to go beyond its size limit, resulting in a compaction to the next level again, and so on. Unlike LevelDB, RocksDB also supports doing compactions in parallel, as long as there are enough available background threads to do so.

2.3.6 Bloom filters

Iterating through every SS-table available to find a single key is inefficient. Instead, we would like to ask the question "can this key possibly exist here?" for each of the SS-tables we go through, and only operate on the ones where the answer is affirmative. With a regular hash-based data structure this would be quite costly in terms of space, as we would need to maintain such a structure for every SS-table in our database. Instead, RocksDB, and many other systems like it, rely on a probabilistic data structure known as a bloom filter [9] to do so.

Instead of knowing with 100% certainty whether a key exists in a set, a bloom filter would let us know if that key *might possibly* be in the set, or if it is *definitely* not. The third option, of *possibly not* being in the set, is impossible. The positive trade-off here is that it uses significantly less space, allowing it to be used for every SS-table in the system.

2.3.7 Iteration

One of the essential features of RocksDB compared to other key-value stores is that its data is *sorted*, and that it can be queried as such through *iterators*. This opens for a wide variety of possibilities that would not have been feasible with a regular key-value store, such as range queries. RocksDB supports iterating both forwards and backwards.

Similar to with reads, performing a fully ordered scan in an LSM-tree storage engine is far from optimal: every tree-structure, or SS-table in RocksDB, needs to be considered, and as key ranges may overlap between different files, sorted.

A lot of applications do not rely on completely random scans of keys however, and only need support for ordered queries within a specific *key prefix*. Developers instruct RocksDB on how to retrieve a specific prefix from each key, which RocksDB then internally uses to organize the data in such a manner that iterating through keys within a *specific prefix* is efficient: either by storing bloom filters for each prefix, or by managing a hash-based index structure based on the prefix.

2.3.8 Column Families

RocksDB supports the equivalent of tables from a traditional database through *column families*[4]. Separate column families share the same write-ahead log but have their own MemTables and SS-tables. Maintaining the same WAL makes it possible to atomically write across multiple column families, while keeping independent LSM-tree components open for the possibility of configuring different column families separately—an important difference from tables in SQL databases.

Column family support was not added until version 3.0 of RocksDB. To maintain backwards compatibility, the default API methods operate on the same column family, "default", with separate methods taking in an additional column family argument.

2.3.9 Customizing the MemTable implementation

RocksDB provides multiple implementations of its in-memory MemTable[5], which can be changed between through *factories*. Different implementations have different advantages and disadvantages, with the default being the all around safest choice.

Skip list

The default implementation uses a *skip list*, a data structure with comparable performance guarantees to a binary search tree—$O(\log n)$ for searches, insertions and deletions—but with far better support for concurrent operations. This makes the default

[4] https://github.com/facebook/rocksdb/wiki/Column-Families
[5] https://github.com/facebook/rocksdb/wiki/MemTable

skip list implementation the only MemTable factory capable of concurrent insertions. Flushing a skip list MemTable to disk is also considerably faster compared to the other factories, with a much lower memory overhead.

Hash skip list

RocksDB provides two hash-based MemTable factories, where keys are organized in buckets based on their extracted *prefix*. This implies that the hash based implementations are only usable when a prefix extractor is defined, and that they only support efficient iterations within a specific prefix. At the same time, the hash-based implementations are also considerably more efficient when that is the case, providing $O(\log k)$ performance, where k is the number of keys within a specific prefix (which is often quite low).

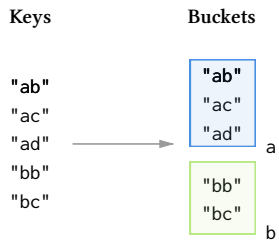

Figure 2.11: In the hash table MemTable implementations, keys are bucketed based on a pre-defined prefix extractor. The figure here uses the first character of each key as its prefix.

Hash linked list

Similar to the skip list based hash table, RocksDB also provides a hash-based implementation where each bucket is maintained as a linked list instead of a skip list. This is similar to a traditional hash table with chaining as its collision resolution, and maintains close to constant time performance guarantees as long as the elements in each bucket is kept low. This comes with significantly lower memory overhead compared to the skip list based hash table but with naturally lower performance when the amount of keys per prefix starts to grow. Because of this, the buckets in a `HashLinkList` are implicitly converted to a skip list when its element count exceeds a certain threshold (256 by default).

Vector

Finally, RocksDB also provides a MemTable factory heavily tuned for random insertions, with abysmal performance for everything else. This makes it only useful for bulk loading data as fast as possible.

2.3.10 Customizing the SS-table implementation

The default SS-table implementation is based on the original format from LevelDB, `BlockBasedTable`[6]. As the name implies, data is stored in separate blocks, where each file's initial block is a filter on the rest of the contents. The size of a single block is usually fixed and can be configured by the application. Read operations always read an entire block into memory, before searching for a specific record. Previously read blocks are maintained in an in-memory cache—a block cache.

RocksDB also provides an improved format designed for low query latency on modern storage media, `PlainTable`[7]. The format was initially developed for in-memory databases, but performs well on other high performance mediums as well. Unlike `BlockBasedTable`, `PlainTable` addresses records by row, and uses a hash-based in-memory index for efficient reads. Similar to the hash-based MemTable formats described in section 2.3.9, it uses *prefix extraction* to place keys into separate buckets. This also limits seek-based iteration to a single prefix.

2.3.11 RocksDB from Rust

While RocksDB is written in C++, it provides a separate API through its C-bindings, which are used to call into it from a variety of different languages[8]. This thesis makes use of a modified version of the `rust-rocksdb`[9] wrapper library, used to call into the RocksDB API from Rust. The majority of the modifications required to make `rust-rocksdb` work with Soup are listed in appendix A.

```
let db = DB::open_default("db_path").unwrap();

let key = b"key";
let value = b"value";
db.put(key, value).unwrap();

match db.get(key) {
  Ok(v) => assert_eq!(*v.unwrap(), value),
  Err(e) => panic!("failed reading from rocksdb: {}", e),
}
```

Listing 2.7: Simple example usage of rust-rocksdb

[6] https://github.com/facebook/rocksdb/wiki/Rocksdb-BlockBasedTable-Format
[7] https://github.com/facebook/rocksdb/wiki/PlainTable-Format
[8] https://github.com/facebook/rocksdb/blob/master/LANGUAGE-BINDINGS.md
[9] https://github.com/spacejam/rust-rocksdb

2.4 Rust

Rust[10] is an open-source systems programming language spearheaded by Mozilla, where it is used to build Servo—a next-generation browser engine[11]. Rust provides memory safety without the runtime overhead of *e.g.*, garbage collection, making it a suitable language for everything from embedded systems to web service backends.

When choosing a programming language, developers are often forced to compromise between higher level abstractions and performance. Large and latency sensitivity projects like databases often opt for the latter, through low-level languages like C. Rust removes this dilemma altogether by providing developers with both the fine-tuned control and performance they are used to in low-level languages, while offering abstractions developers might be familiar with from interpreted languages.

```
fn suffix(input: &mut String) {
    input.push_str(" is a String!");
}

fn output(input: String) {
    println!("Hello: {}", input);
}

fn main() {
    // Construct a mutable String, from a string literal (str):
    let mut input = String::from("Hi!");
    // Pass a mutable reference to suffix:
    suffix(&mut input);
    // Finally, move our input variable into the output function:
    output(input);
    // println!("This line would not compile: {}", input);
}
```

Listing 2.8: The example shows the basics of Rust's move semantics. The `input` variable cannot be used after the call to `output()`, as it has been *moved* into the function.

One of Rust's key features is providing compile-time safety both in terms of types and memory. The latter is done through an ownership model which lets developers program mostly without thinking about memory allocation and deallocation, without the lowered performance of using something like a garbage collector. Each variable in Rust is assigned one and only one owner, and the variable is deallocated when that owner goes out of scope.

[10] https://www.rust-lang.org/
[11] https://servo.org/

2.4.1 Foreign Function Interface

Rust has excellent support for calling into C programs, which lets developers access the myriad of libraries written in C, together with C++ programs that provide C interfaces. To call external programs, developers define each external function in a *foreign function interface*[12], through use of the `extern` keyword, which also supports linking to external libraries.

```
extern "C" {
  fn rand() -> i32;
}

fn main() {
  unsafe {
    println!("Random number: {}", rand());
  }
}
```

Listing 2.9: By defining `rand` from the C standard library as an external function, we can call it from our Rust program.

Note that the call to `rand` in listing 2.9 needs to be wrapped in an `unsafe` block. While Rust can ensure the safety of *Rust* code at compile time, it cannot not do so for third-party applications written in other languages. By introducing the `unsafe` keyword, the blocks of code the developer is forced to maintain the safety of is isolated to the smallest possible region.

In the same manner, Rust supports defining an interface that can be called from other languages, such as C, as shown in listing 2.10. This is especially useful for libraries that require functions as arguments, *e.g.*, callbacks.

```
#[no_mangle]
pub extern "C" fn multiply(a: i32, b: i32) -> i32 {
  a * b
}
```

Listing 2.10: The `multiply` functions can be called through the C-calling convention by other programs. The `no_mangle` pragma ensures the `multiply` name stays unmodified by the compiler.

[12] https://doc.rust-lang.org/book/first-edition/ffi.html

2.5 bincode

`bincode` [63] is a binary serialization library, used heavily throughout both this thesis and in Soup, for everything from RPC communication to persisting data to durable storage. In short, `bincode` takes an arbitrary Rust object and turns it into a series of bytes—an encoded object. The size of the resulting byte stream is usually either less than, or the same as, the size of the source object. `bincode` builds on top of the `Serde`[13] serialization framework.

As the encoded format is of relevance to later sections, we will briefly go through it here. Primitive values, such as numbers, are encoded directly using Rust's `Writer` trait, with a few exceptions:

- `isize` and `usize` types, which have varying sizes depending on the OS, are encoded as `i32` and `u64` correspondingly.
- Strings are encoded as the tuple `(number of bytes, bytes)`, where the former is a `u64` and the latter is a byte slice.

Compound types—enums, structs, vectors, and tuples—are encoded recursively, with each of their fields placed out in succession. With vector lengths not being determined at compile time, vectors are prefixed with a length field on the form of a `u64`. This is not necessary for the other compound types, as their sizes do not vary at runtime. An enum instance can represent multiple types, and is prefixed with a `u32` tag used to determine which enum variant it represents.

[13] https://serde.rs/

```rust
#[derive(Serialize, Deserialize, Debug, PartialEq)]
enum Number {
    Positive(u64),
    Negative(u64)
}

fn main() {
    let values = vec![Number::Positive(3), Number::Negative(4)];
    // This serializes as:
    // vector length u64,
    // + enum variant u32 + u64,
    // + enum variant u32 + u64
    // = u64, u32, u64, u32, u64
    // = 32 bytes
    let raw = bincode::serialize(&values).unwrap();
    let deserialized: Vec<_> = bincode::deserialize(&raw).unwrap();

    for (i, element) in values.into_iter().enumerate() {
        assert_eq!(element, deserialized[i]);
    }
}
```

Listing 2.11: Types implementing the `Serialize` and `Deserialize` traits can be encoded and decoded using `bincode`.

2.6 Profiling

This section describes various tools used to reason about performance bottlenecks throughout the thesis.

2.6.1 CPU

A large part of application performance tuning comes down to figuring out which portion of a program is running slowly and why that is the case. Throughout this thesis that is accomplished using `perf`[14]—a profiling tool that helps us answer the question "What is the CPU spending time on?".

`perf` collects information from both hardware counters and logical tracepoints. The latter is especially useful for recording call graphs of a program, which in turn lets us produce flame graphs like the one in figure 2.12, using tools such as `FlameGraph`[15] and

[14] https://perf.wiki.kernel.org
[15] https://github.com/brendangregg/FlameGraph

`FlameScope`[16]. Flame graphs show time spent on the horizontal axis, while showing the call graph vertically.

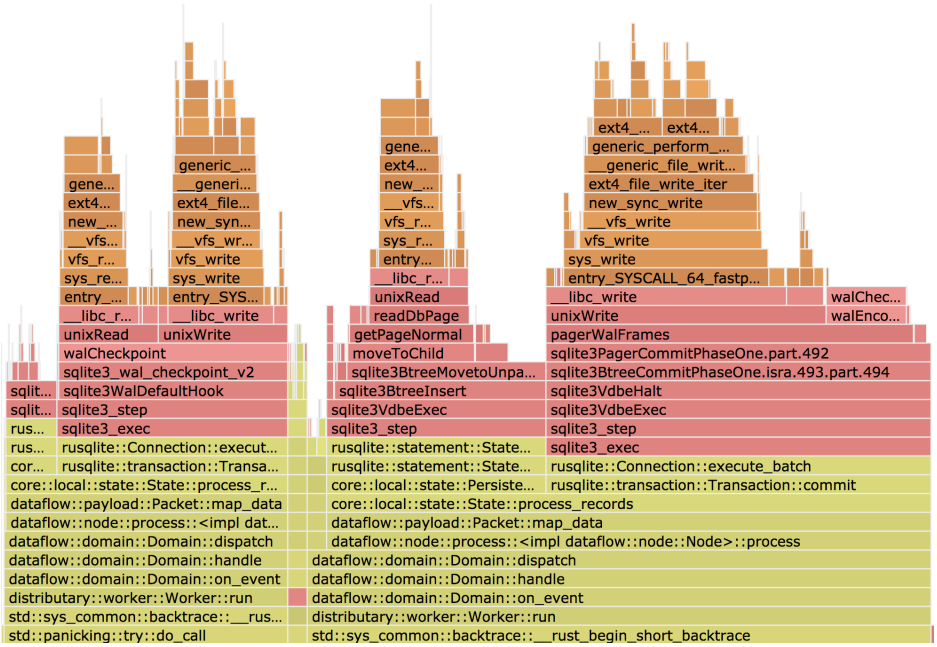

Figure 2.12: An example flame graph from events recorded using `perf`.

2.6.2 Memory

Memory leaks become an issue when we continually allocate memory without freeing it. This can easily happen in languages without dynamic memory allocation, when a programmer forgets to deallocate some portion of memory after using it. Not to say that it is non-existent in garbage collected languages, where it instead often correlates to logical errors, *e.g.*, by continuously attaching listener functions to an event system without regard for previous subscriptions. Rust's ownership system largely prevents issues of the first kind from happening—variables that go out of scope are deallocated automatically. Regardless, Rust supports calling into arbitrary C-programs (see section 2.4.1), where memory can be allocated and deallocated without regard for safety.

To profile memory leaks, we use the Valgrind Massif[17] heap profiler. Massif continuously takes snapshots of the heap, recording what memory is used for, and where that memory was allocated from. While this is subject to change in the future, the current version of Rust, version `1.26.2`, uses the `jemalloc`[18] memory allocator instead of the system's

[16] https://github.com/Netflix/flamescope
[17] http://valgrind.org/docs/manual/ms-manual.html
[18] http://jemalloc.net/

default allocator. Unfortunately, memory profiling using Valgrind does not work well with `jemalloc`. To resolve this we can instead force Rust to use the system allocator, at least while profiling.

```
#![feature(alloc_system)]
#![feature(global_allocator, allocator_api)]

#[global_allocator]
static ALLOC: std::alloc::System = std::alloc::System;
```

Listing 2.12: Forcing Rust to use the system memory allocator makes it possible to profile it using tools such as Valgrind Massif.

Chapter 3

Related work

This section reviews ideas relevant to the main contributions throughout this thesis, both from existing research and from various industry implementations. The first section investigates concepts crucial to building secondary indexing schemes on top of key-value stores, implemented in chapter 5. The second section looks at recovery solutions for main-memory databases, followed by a dive into snapshotting techniques in distributed systems, necessary for chapter 6.

3.1 Indexing

Index structures are used by databases to facilitate efficient retrieval. While a majority of traditional database systems maintain indices separate from the data itself (which could be stored in *e.g.*, a heap file [57, 75]), it has become increasingly common to co-locate rows with the index—often referred to as a clustered index. Systems such as InnoDB [59] and comdb2 [70] rely heavily on B-trees for both indexing and row storage, achieving overall decent read performance on a wide variety of storage mediums.

At the same time, a gradual increase in write-intensive applications have resulted in a myriad of log-structured merge tree based storage systems—a data-structure which usually requires less write amplification than B-trees [10]. While LSM-tree based systems (*e.g.*, Google Bigtable [12] and Apache HBase [78]) provide excellent availability and scalability, their key-value based APIs are restrictive, and lack features such as secondary indexing.

Key-value APIs are sufficient for many applications, while others require more advanced features. Google Spanner [19] and its open-source competitors remove the need to compromise between strong consistency and scalability, and provide an SQL-based query interface to its users. While Spanner combines an implementation based on Bigtable with Paxos [51] to provide distributed consistency, CockroachDB [14] and TiDB [65]—both open-source—do the same with the LSM-tree key-value store RocksDB [29] and the Raft consensus algorithm [62].

Both CockroachDB and TiDB implement advanced features (*e.g.*, replication and sharding) as layered abstractions, with RocksDB's ordered key-value API at the core. With clever key schemes and heavy use of RocksDB's iteration properties, CockroachDB and TiDB can support secondary indices on top of RocksDB—a well-supported and heavily tested library with reliable performance guarantees. Other projects, such as SLIK [46], HyperDex [26], and Replex [76], implement secondary indexing as first-class citizens in new distributed key-value stores built from the ground up.

3.1.1 Secondary indices with LSM-trees

LSM-tree systems achieve high write throughput in part by buffering updates in memory, amortizing the disk write penalty across a batch of writes. AsterixDB [3] recognizes the effectiveness of the LSM-tree approach, and applies the same technique to in-place update index structures. The process—which they refer to as LSM-ification—lets AsterixDB build secondary indices using read-optimized data structures, *e.g.*, B-trees. While this is an interesting approach, on-disk data structures are far from trivial to implement, and additional data structures undoubtedly increase a system's overall complexity—even if the data structures are built on the same components.

Another approach is to build index schemes on top of the existing APIs provided by LSM-trees, notably the `Get`, `Put`, `Delete`, and `Seek` operations provided by systems such as RocksDB. In [66], LSM-tree index structures are split into two categories: *standalone*

and *embedded*. The former maintains secondary indices in separate key-spaces, while the latter embeds the necessary information without additional stored rows.

Standalone indices

In traditional relational database management systems, indices are usually maintained as separate data structures (*e.g.*, B-trees) with pointers to primary key values. Standalone indexing in [66] stores pointers as well, either in the same key-space as regular updates or in separate tables (often referred to as *column families* in LSM-based systems). These pointers can be maintained either *eagerly* or *lazily*, *e.g.*, either synchronously on the main-path or asynchronously in the background. The latter introduces complexity to the system, but offers potentially improved insertion and update performance in return [18, 77].

How the secondary index pointers are structured varies from scheme to scheme. The perhaps most obvious way of doing so is to maintain a serialized list of primary keys for each secondary index key, which is then retrieved and updated on each insertion. This is referred to as a *posting list* in [66], and *table-based* secondary indexing in [23]. In the latter, where a secondary index scheme is implemented on top of HBase [78], a serialized `TreeSet` is maintained for each secondary index value. The `TreeSet` is updated synchronously on new insertions, which [66] refers to as *eager indexing*. The alternative, *lazy indexing*, would instead issue only an insertion for new values, and take care of the concatenation either in the background or during read operations (*merge operator* in RocksDB [28]). While the lazy alternative offers far greater insert performance by avoiding random reads, it still requires potentially costly list serialization and deserialization.

id	brand	color
1	volvo	silver
2	volvo	blue
3	audi	red

key	value
volvo	[1, 2]
audi	[3]

Table 3.1: A separate list of primary keys is maintained for each secondary index key.

Another alternative is to rely on the ordered iteration properties available in LSM-tree based systems such as LevelDB [39] and RocksDB [27]. By suffixing secondary keys with unique primary keys (composite keys in [66]), the pointers can be retrieved by iterating through all keys that start with a given secondary index prefix, removing the need to store anything in the value portion at all. While this requires care to make sure that values with the same prefix are ordered next to each other, it completely removes the need for random reads when inserting new values. This is similar to how systems such as Spanner [6], TiDB [71], and CockroachDB [15] implement secondary indices.

id	brand	color
1	volvo	silver
2	volvo	blue
3	audi	red

key	value
volvo-1	
volvo-2	
audi-3	

Table 3.2: Secondary index keys are suffixed with the primary key they point to, and can be retrieved by iterating through all secondary index rows with the correct prefix.

Embedded indices

Instead of storing separate index pointers for secondary indices, [66] presents an alternative where bloom filters are used to determine whether an on-disk block contains rows with a given secondary index attribute or not. Queries then iterate through all blocks, referring to the in-memory bloom filter to determine whether it requires scanning for potential rows. To retrieve values from the LSM-tree memory buffer, a separate B-tree is maintained in-memory for each secondary attribute.

Embedded indexing reduces write-amplification when inserting new rows—no extra index data needs to be persisted to disk. In turn, it reduces read performance, as retrievals now need to consider every block available, even if only ends up reading a small subset.

3.2 Recovery

Database researchers observed early on that users needed a way of performing a series of operations as a unit, where the result would either be made available to concurrent users as one, or not at all—a transaction [8]. At the same time, failures are inevitable in any system, and ensuring that the result of previously *committed* transactions still remained after crashing was crucial. Together, these requirements formed a subset of the ACID [40] principles (atomicity, consistency, isolation, and durability).

ARIES—Algorithms for Recovery and Isolation Exploiting Semantics [58]—has in-large remained the gold standard in transaction recovery algorithms for three decades. ARIES persists all changes—regardless of commit status—to a durable write-ahead log. During recovery, ARIES first applies all missing updates from the log, before it finally reverts changes belonging to uncommitted transactions. The former, `REDO`, maintains durability, while the latter, `UNDO`, upholds atomicity. By sequentially persisting all changes to the log, ARIES systems are free to write dirty pages to durable storage at any point, and does not need to do so prior to committing. Referred to as correspondingly *steal* and *no-force*, this allows for high throughput processing at the price of increasing complexity.

While the logging structure varies from implementation to implementation, the principle of a write-ahead log remains the same. By appending changes to a persistent log prior

to updating index structures, we avoid the performance penalties of random writes to durable storage, while still ensuring durability in the face of a potential crash. To maintain atomicity for transactions, we also log enough information to either safely revert their changes, or fully persist them after recovering. With the introduction of fast non-volatile memory, the age old wisdom of preferring sequential writes over random updates might slowly go away [5, 13]. Regardless, to build systems that perform well on hardware most users have access to—still in-large spinning and solid-state drives—the arguments in-favor of write-ahead logging still remain.

3.2.1 Recovery in main-memory databases

Traditional relational database systems were never built with the goal of storing entire datasets in main-memory. B-trees, concurrency control techniques, buffer pools, and other components were instead built with the opposite in mind—efficient processing of data residing on slow durable storage mediums. Today's cheap access to vast amounts of volatile main-memory requires different thinking, which has given rise to a new type of structured storage system: main-memory databases [21].

According to [41], the SHORE[1] database system spends over 10% of its processing time maintaining an ARIES-style log. For main-memory systems capable of processing thousands of transactions per second, the penalty of writing to durable storage would be far beyond 10%. Regardless, main-memory database systems still need to ensure durability somehow, otherwise they would merely be large data structures. VoltDB introduced the concept of *command logging* [54], where the operation performed is logged instead of the results of its modifications.

Logging logical operations, *e.g.*, SQL queries, reduces the amount of data written to durable storage. Whereas an ARIES-style log would have to write the results of the operations performed, a command log would only need to persist the intent of the operation itself. Maintaining a command log reduces the computational overhead of durability, by removing the need to calculate before and after images of modifications. VoltDB, HyPer [47], and Hekaton [22] group operations from multiple transactions together in a single batch before writing to durable storage, amortizing the fixed cost of syncing to persistent storage across multiple transactions. Since main-memory systems never write dirty pages to disk, they do not have to make sure that log entries are written prior to committing, as a failure would not require undoing changes on durable storage. Instead, they have to delay write acknowledgments until the entire batch has been persisted. Batching updates with a *group commit* scheme greatly increases a system's write throughput, at the expense of potentially increased latency of individual operations.

Command logging does on the other hand increase the effort needed to recover from a failure. Whereas an ARIES-style log contains the computed results of each operation, a command log does not, and needs to redo potentially costly computations while recovering. VoltDB, however, claims that failures are rare, and that the focus should

[1] http://research.cs.wisc.edu/shore-mt/

be on reducing run-time overhead, even if it comes at the cost of increased recovery latency. Regardless, recovering from an ever-growing log of entries is not a feasible choice. Applications with consistently high throughput would never be able to recover, as they would have to redo all write operations previously performed by the system.

To avoid an infinitely growing write-ahead log, main-memory databases *checkpoint* their state at regular intervals [21, 22, 24, 44, 54]. While far from a new concept, increasing performance demands have led to the development of more sophisticated checkpoint methods [52, 67]. The implementation details vary, but most systems agree that checkpoints should be performed in an asynchronous manner, without blocking the system, and without inducing a significant performance penalty. From this, [67] defines a few key properties:

1. Checkpointing should not significantly slow down transactional throughput.
2. Checkpointing should not drastically increase regular processing latency.
3. Checkpointing should require as little extra memory as possible.

While the overhead of an algorithm following the aforementioned properties is minimal, the asynchronous checkpoint method presented in [67] still introduces a window of time where throughput degrades with 10%. To completely avoid the overhead of durable storage, main-memory databases like H-Store rely on replication for durability [45]. By replicating data to $K+1$ nodes, the system maintains transactional durability for up to K failures. *K-safety* [74] is however naturally susceptible to total failures, *e.g.*, in the event that multiple data centers fail. As a compromise, some systems implement replication together with checkpointing, limiting the amount of data lost in a total failure.

3.2.2 Snapshotting in distributed systems

Determining the global state of a distributed system is a useful property in scenarios ranging from steady-state detection of deadlocks, monitoring, debugging, and finally, failure recovery [49]—the aspect of which snapshotting is used for in this thesis. By combining local checkpoints across the Soup data-flow graph, a global checkpoint—a snapshot—can be formed, and later recovered from in case of failures.

To perform a global snapshot, each node needs to record their local state at the same instant across the system, without sharing memory or access to a global clock. At the same time, the snapshot should happen without pausing regular processing. Chandy and Lamport first introduced the problem of acquiring a distributed snapshot in [11], which has since been the source of inspiration for a wide variety of work within the field. Chandy and Lamport presented a solution aimed at distributed systems using first-in first-out channels, with preserved message ordering, by solving two main issues: deciding when to take a snapshot, and which messages should be part of said snapshot.

> "*The state-detection algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds—a scene so vast that it cannot be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. The snapshots cannot all be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed; for instance, they cannot get all the birds in the heavens to remain motionless while the photographs are taken. Yet, the composite picture should be meaningful. The problem before us is to define "meaningful" and then to determine how the photographs should be taken.*"
>
> Chandy and Lamport's description of the global snapshotting problem.

Chandy-Lamport's key insight was to introduce a marker message, used as a separator between messages that should be included in the snapshot and messages that should not. Processes that receive a snapshot marker should immediately take a snapshot of all messages received prior to the marker, and forward the resulting state to a process capable of assembling all its received local snapshots to a global view of the system. The channels' FIFO property ensures the exclusion of messages arriving after the marker. The resulting algorithm requires $O(e)$ messages to initiate a snapshot, where e is the amount of edges in the graph. The messages can be sent out in parallel, resulting in a $O(d)$ guarantee to complete the snapshot, where d is the diameter of the graph.

Later on, Spezialetti and Kearns improved the Chandy-Lamport algorithm by recognizing that the combining phase of snapshotting could be performed concurrently across the graph [56]. Instead of having all nodes send a snapshot to a single initiator, each node in the graph would be assigned a *parent* which it would forward its local state to, forming a spanning tree of initiators across the graph.

Chandy and Lamport's algorithm is designed with FIFO channels in mind, and a second class of snapshotting algorithms extend Chandy-Lamport in various ways for non-FIFO channels. Lai and Yang [50] introduced a solution that removes the need for explicit control messages, by including the required snapshotting information in existing messages. To maintain consistency without explicit marker messages, Lai-Yang makes use of a two-coloring scheme. Every process is initially white and turns red when initiating a snapshot, while recording all messages since the last snapshot was taken. Another alternative is the Mattern algorithm [55], which makes use of vector clocks to perform global snapshots in non-FIFO environments.

Systems that uphold causal ordering of all messages open for a third category of snapshotting algorithms. A causally ordered system guarantees that for two messages m_1 and m_2, if $sent(m_1) < send(m_2)$ then $deliver(m_1) < deliver(m_2)$ for any common destinations of both m_1 and m_2. This removes the need for Chandy-Lamport's explicit synchronization markers, and both Acharya-Badrinath [1] and Alagar-Venkatesan [2] present snapshotting algorithms where the initiator requests a snapshot directly from each node, reducing the required messages from $O(e)$ to $O(n)$.

Chapter 4

Benchmarks

Soup includes a series of benchmarks designed to reproduce different real world scenarios where usage of Soup might be appropriate. The Lobsters and Vote benchmarks existed prior to this thesis, while the Recovery and Replay benchmarks were explicitly built to highlight the positive and negative impact the features developed in this thesis have on Soup.

4.1 Hardware

This section describes the various servers used to generate the results in chapter 7.

4.1.1 Server setup 1: SSD

Unless otherwise specified, benchmarks are run on Dell PowerEdge R430 server with two Intel Xeon E5-2660 v3 CPUs and a total of 20 physical and 40 logical cores. The server has 64GB of DDR4 RAM running at a speed of 2200MHz, with two solid-state drives: a Samsung SSD 850 PRO and an Intel SSD S3710.

4.1.2 Server setup 2: EC2 NVMe SSD

Some of the benchmarks require the workload generator and clients to be separated on a different machine than Soup itself. For these cases, the benchmarks are run on Amazon's Elastic Compute Cloud (EC2) instances[1], typically with the workload generator running on a machine with a large number of cores and the Soup server itself running on a server with fewer and faster cores.

When Soup needs access to fast durable storage, an `m4.10xlarge` instance is used for the clients and an `i3.4xlarge` instance is used for the Soup workers. The `m4.10xlarge` server uses an Intel Xeon E5-2686 CPU with 40 logical cores and 160 GB of RAM. The `i3.4xlarge` uses the same CPU as the `m4`, but is backed by NVMe SSDs capable of extremely high throughput I/O.

4.1.3 Server setup 3: EC2 RAM Disk

Similar to the setup in the previous section, the third setup makes use of two servers hosted on Amazon EC2—this time without fast durable storage. Instead, an `m5.12xlarge` and a `c5.4xlarge` is used. Both make use of newer Intel Xeon Platinum processors, with 48 cores and 192GB RAM on the `m5` and 16 cores and 32GB RAM on the `c5`.

4.2 Lobsters

Lobsters[2] is a news aggregation website where users post, vote, and comment on links and discussions. Soup uses Lobsters to showcase the performance advantages of Soup in a real-world application. While Lobsters is built with Ruby on Rails, the Soup benchmark runs MySQL queries normally issued by Lobsters directly against Soup, using the MySQL protocol shim described in section 2.1.6. This avoids the overhead of Ruby and Ruby on

[1] https://aws.amazon.com/ec2/instance-types/
[2] http://lobste.rs/

Rails, which quickly become bottlenecks when the Lobsters traffic is scaled beyond its regular workload.

Whereas the other benchmarks focus on individual writes and reads, the Lobsters benchmark is measured in page views, where different pages execute a series of write and read queries. The distribution of page views is modeled after real Lobsters production traffic, to ensure the queries executed best resemble a real Lobsters setup.

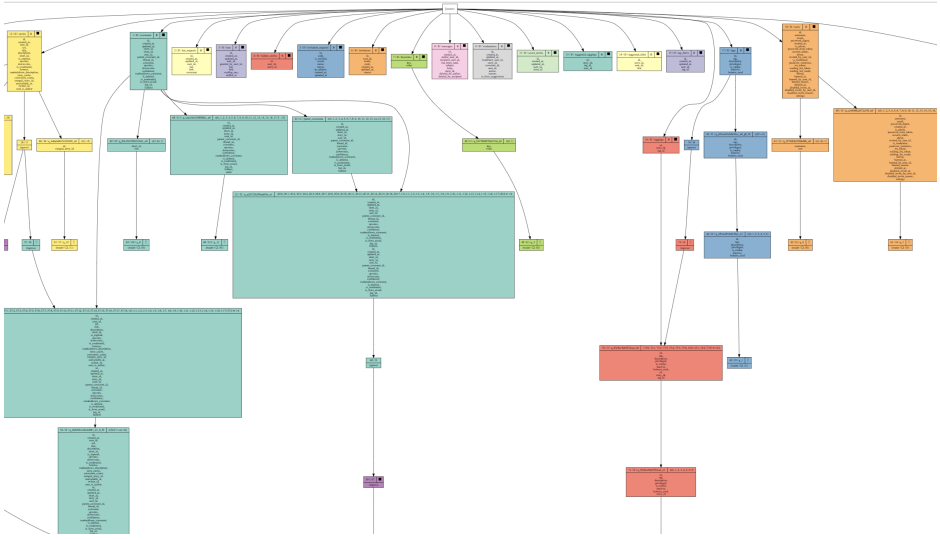

Figure 4.1: A subset of the Soup data-flow graph used to run the Lobsters benchmark.

The regular Lobsters queries rely on manual materializations and other optimizations to reach decent performance using MySQL. Part of Soup's goal is to let developers write "natural" queries, without caching or other denormalizing optimizations, and some of the Lobsters queries have been rewritten to better highlight this.

4.3 Vote

The Lobsters benchmark includes a wide variety of different queries, making it difficult to narrow down bottlenecks in Soup's query processing performance. The vote benchmark solves this by focusing on the most frequently run query in Lobsters, reading stories and their corresponding vote counts.

```
CREATE TABLE Article (id int, title varchar(255), PRIMARY KEY(id));
CREATE TABLE Vote (article_id int, user int);

QUERY ArticleWithVoteCount:
  SELECT Article.id, title, VoteCount.votes AS votes
  FROM Article
  LEFT JOIN (
    SELECT Vote.article_id, COUNT(user) AS votes
    FROM Vote
    GROUP BY Vote.article_id
  ) AS VoteCount
  ON (Article.id = VoteCount.article_id) WHERE Article.id = ?;
```

Listing 4.1: The schema used by the vote benchmark.

The database is prepopulated with a series of articles, letting the write portion of the benchmark focus on writing votes, where each vote is assigned to an article following a uniform distribution. The vote benchmark runs either locally against a single Soup instance, or against a cluster of Soup workers.

4.3.1 Open-loop

In a closed-loop benchmark, a new request is issued when the previous completes. With open-loop, requests are instead issued independently, resulting in a model more closely resembling a real-world scenario [69]. Soup's vote benchmark relies on a *partially* open-loop setup, where load is generated by clients based on a specified distribution, while maintaining a capped queue of outstanding requests. This prevents reductions in measurements during slower processing periods, where a closed-loop benchmark would issue far fewer requests than an open-loop one.

Two latency measures are recorded during the vote benchmark: the time it takes for a single batch to be processed (*sojourn* time), and the time it takes from the request is generated to it completes (*batch processing* time). The former is usually higher than the latter, as it includes the delay from the request is queued until it is processed by the system.

4.4 Replay

Part of the promise of Soup is to avoid expensive computations for reads, by moving most of the workload to the write portion of the system. Read queries trigger replays for keys initially, while later requests are served directly by partially materialized state further down the graph. This makes it difficult to reason about the read performance of

Soup's base nodes, as only a small portion of reads are bound to be served by the base nodes at all.

This is where the replay benchmark comes in. Instead of possibly reading the same keys multiple times, the replay benchmark ensures that each key is only ever read *once*, triggering a replay from the base nodes. Additionally, the schema is far simpler than in the vote benchmark, with a single base node and different variants of the same query. This avoids excessive computation in partial nodes in the lower portions of the graph, and focuses the main portion of the benchmark on state processing at the base node level.

```
CREATE TABLE TableRow (
  id int, c1 int, c2 int, c3 int, c4 int,
  c5 int, c6 int, c7 int, c8 int, c9 int,
  PRIMARY KEY(id)
);

/* Primary key reads: */
QUERY ReadRow: SELECT * FROM TableRow WHERE id = ?;

/* Secondary key reads */
QUERY query_c1: SELECT * FROM TableRow WHERE c1 = ?;
/* .. */
QUERY query_c9: SELECT * FROM TableRow WHERE c9 = ?;
```

Listing 4.2: The schema used by the replay benchmark.

While trifling for the regular Soup base node implementation, the difference between reading from a primary and secondary index might be consequential with persistent base nodes (see chapter 5). To correctly assess this difference, the replay benchmark includes the possibility of reading either from a primary or a secondary index. The row size is also slightly larger than in the other benchmarks, ensuring that the base nodes actually have to read a fair share of data from state.

Prior to reading, Soup is populated with a given number of rows, which it then reads a uniform, and smaller, sample of. With persistent base nodes, the benchmark terminates after prepopulation, followed by a recovery step, to ensure that data is served directly from durable storage. The file system cache is also cleared after recovery and between subsequent benchmark runs.

4.5 Recovery

The final benchmark measures the recovery time of our voting application, using the schema described in listing 4.1. The database is first populated with a series of articles

and votes, before being shut down and recovered from. The benchmark produces two measures: initial and total recovery time. The former records how long it takes for the first read to return correct data, while the latter only finishes when up-to-date data is returned from *all* keys.

Chapter 5

Persistent base tables

Updates begin their journey through the Soup data-flow graph at the base nodes, after being successfully persisted to Soup's write-ahead log. While nodes further down in the graph might be *partial*, the base nodes always contain every single record stored in a Soup application. This is crucial in maintaining a balance between efficient read queries and space usage: popular queries will be handled by partial state further down the graph, while reads for infrequently accessed rows will be able to refer all the way up to the source of truth, the base nodes, through what in Soup is called a replay. In comparison to existing database systems, base nodes are closest to what otherwise might be known as *tables*.

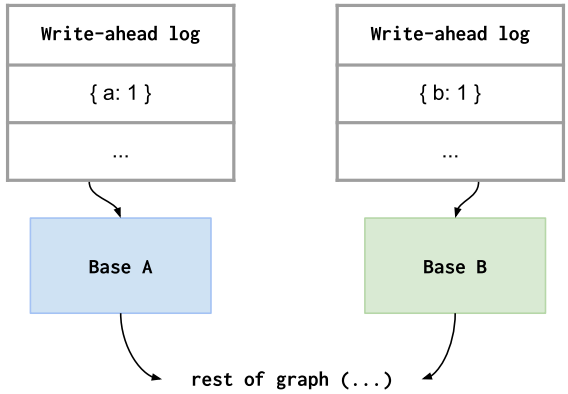

Figure 5.1: Updates enter the base nodes after being persisted to a write-ahead log.

While partial nodes can use *eviction* (see section 2.1.1) to keep their memory footprint

low, the size of the base nodes will continue to grow unbounded throughout a Soup instance's lifetime. In the short term this can be handled by sharding Soup's data across multiple machines in a cluster, however, this is infeasible in the long term: sustained write workloads would continue to grow the base node state, regardless of whether the data is accessed by queries or not.

To combat this, we would like to move either part or all of the state stored in base nodes to durable storage. This would reduce Soup's overall memory usage, and perhaps even more significantly, transition Soup from a purely in-memory database to a system that can store more data than its available memory. With data safely persisted to the base nodes, recovery after a failure would also require less work, as partial nodes could gradually recover when data is requested through replays. Summarized, introducing persistence to Soup's base nodes would achieve the following goals:

1. Prevent Soup's memory usage from growing unbounded over time
2. Support larger-than-memory data sets
3. Reduce recovery time after failures

5.1 In-memory state

The current in-memory state implementation provides a key-value API with support for multiple indices. A separate state data structure is kept for each materialized node, including base nodes. The same data structure is used by both partially and fully materialized nodes, but as base nodes always have to be fully materialized, this section will omit describing details regarding the former. The state data structure will be referred to as `State`.

5.1.1 Adding indices

The `State::add_key` method introduces a new index to a specific `State` map, and takes a set of columns as its argument. Additional indices do not lead to multiple copies of the data, but rather contain pointers to the existing rows. These pointers are held in separate data structures: each new index introduces a new hash map structure responsible for answering queries for that specific set of columns. New indices have to be *built*, as they are expected to answer queries for data that has already been inserted, right away.

```
state = State()
// Initialize `state` with an index on the first column:
state.add_key([0])

// Then insert two rows:
state.insert([A, 1])
state.insert([B, 1])
assert_equal(state.lookup(A), [[A, 1]])

// Now, add an index on the second column:
state.add_key([1])
// ...which should return values that existed
// prior to the index being added:
assert_equal(state.lookup(1), [[A, 1], [B, 1]])
```

Listing 5.1: Pseudo-code test that shows the expected behavior for adding indices with existing values.

5.1.2 Operations

Retrieval

A stateful map would not be useful without a method for retrieving data, which `State` provides through `State::lookup`. This takes a set of columns and a key as its argu-

ments, and make use of a single index to retrieve one or more existing values. Each index is held in a separate hash map, and retrievals can thus be completed in constant time.

The fact that `State::lookup` can return more than one value is an important distinction from most key-value stores, and is absolutely crucial in implementing secondary indices. Without it, `State` would only be able to serve as an index structure for *unique primary keys*.

Insertion

Naturally, to be able to read anything in the first place, we first need to insert data into our stateful data structure. This is done through the `State::insert` method, which takes a single row as argument. `State::insert` is responsible for updating every index in `State`, and has to loop through and insert pointers for each included index.

Removal

Similar to insertions, removals have to update all indices. Contrary to what one might expect, `State::remove` takes a single row as argument, and not a key. This is due to how `State` is used in Soup: as an internal storage unit for partially and fully materialized nodes alike. While it might make sense for a base node to only delete rows by key, other nodes need support for deleting a single row, regardless of if its key is unique or not. No matter, translating `remove(row)` to `remove(key)` is trivial, as shown in listing 5.2.

```
state = State()
state.add_key([0])
state.insert([A, 1])

row = state.lookup(A)
state.remove(row)
```

Listing 5.2: Deleting a row from a base node in Soup.

Updates

Similar to how `State` does not have a method for removing by key, `State` does not include a method for updating a single row. Updates are instead handled by the base node's logic, by first emitting a negative record—to delete the existing row—followed by a positive one to insert the new value.

5.2 Requirements

Random access memory is, and has been for years now, fast. Durable storage is undergoing a similar transformation with the recent introduction of NVMe SSDs [81], but is still orders of magnitudes slower than RAM. This means that while introducing persistent storage into parts of the Soup equation comes with many benefits, increased performance is not likely to be one of them. Rather, the goal will be to maintain the existing performance characteristics, while still achieving our three main goals.

5.2.1 Write throughput

Backfills of missing state, replays, go through the regular update paths in the Soup data-flow graph. An increase in processing latency at the base nodes would not only reduce the write throughput, but also have a significant impact on reads that require a replay to complete.

5.2.2 Point query performance

Base node queries should be a seldom occurrence, as most queries should be served by nodes further down the graph. Still, when a query eventually makes it all the way up to the base nodes, it is important that it can be completed fast enough to not significantly slow down the total read throughput of the system. Slightly higher latency is on the other hand to be expected, as we are after all comparing persistent storage to an in-memory hash map.

5.2.3 Support both primary and secondary indices

Similar to Soup's existing `State` implementation, a durable replacement has to support mapping a single key to multiple values, *i.e.*, secondary indices.

5.3 Embedding an existing storage engine

On-disk data structures have wildly varying performance characteristics. A B+tree (see section 2.2.1) might perform well on random reads, but get heavily out-performed by an LSM-tree (see section 2.3) on sequential writes. At the same time, decades of changes in hardware research have broadened the field further. A new NVMe SSD is able to reach almost half a million random reads per second[1], whereas a traditional spinning

[1] `http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/`

disk barely scratches the surface of a hundred[2]. This makes building data structures for durable storage non-trivial and time consuming.

On the other hand, there is a plethora of existing, open-source, storage backends available today. Similar to their underlying data structures, different backends are built for different use cases, with different hardware in mind. This provides an option to implementing data structures from scratch, by instead making use of existing database systems to test performance assumptions, which is exactly what this thesis will do: first using SQLite, and later using RocksDB.

5.3.1 State interface

Before diving into the individual `State` operations, we need to pave the way for the possibility of even having two different state implementations: the existing in-memory implementation, and the new persistent storage variant, from here on referred to as `MemoryState` and `PersistentState`. This is achieved by turning the existing `State` implementation into an interface—a *trait* in Rust—which would then be implemented by both the `State` variants. This helps maintain the current status quo where `State` is an *abstract data type*: internal Soup callers do not need to be aware of the location their data is getting stored—they can simply interact with the `State` trait as a black box.

```
pub trait State {
    /// Add an index keyed by the given columns.
    fn add_key(&mut self, columns: &[usize]);

    /// Inserts or removes each record into State
    fn process_records(&mut self, records: &Records);

    /// Retrieve values from the index defined for `columns`.
    fn lookup<'a>(
      &'a self,
      columns: &[usize],
      key: &KeyType
    ) -> LookupResult<'a>;

    /// Count the rows currently stored in `State`.
    fn rows(&self) -> usize;

    /// Return a copy of all records.
    fn cloned_records(&self) -> Vec<Vec<DataType>>;
}
```

Listing 5.3: A segment of the main methods defined in our `State` trait.

[2] https://www.symantec.com/connect/articles/getting-hang-iops-v13

As code is often more succinct than prose, a subset of the `State` trait is shown in listing 5.3. The rest of the methods have been omitted for clarity, as they are only relevant to nodes that can be partially materialized—not base nodes. Similarly, some of the methods take in extra arguments related to partial state—omitted here.

Comparing the trait in listing 5.3 to the operations described in 5.1 one might notice that the insert and removal operations are gone. These have been abstracted into a higher level method: `process_records`. Every packet in Soup has the potential to contain more than one record by being a merged packet, due to *group commit* (see section 3.2.1). Because of this, the function responsible for materializing records in a node's `State` would go through a packet's records, individually calling methods like `State::insert` and `State::remove`. This is completely valid for an in-memory implementation, where each operation has the same cost—without any initial overhead. This is not the case for writes to potentially slower, durable storage. Here, batching is key, and an indicator to the underlying methods that they can perform operations in one go is crucial.

5.3.2 Ownership of data from `State`

While other languages might implement memory safety through garbage collection or manual memory management, Rust does the same through ownership, as described in 2.4. Whenever a value goes out of scope, it is deallocated. In a garbage collected system, this happens when there are no longer any references to the value. In Rust, each value only has one owner, and any references need to live **at least** as long as the value created by that owner.

On the other hand, values sometimes need to be owned by more than one location. This is often the case for data structures, and `State` is no exception here. Values stored in `MemoryState` should not have to be cloned during retrieval, which would incur a heavy performance penalty. Instead, retrieving values from `State` return dynamically reference counted[3] values, allowing shared ownership of a single value by counting owners at runtime. Subsequent retrievals of the same value always point to the same memory location, with the source of truth being stored in `State`.

[3] `https://doc.rust-lang.org/std/rc/index.html`

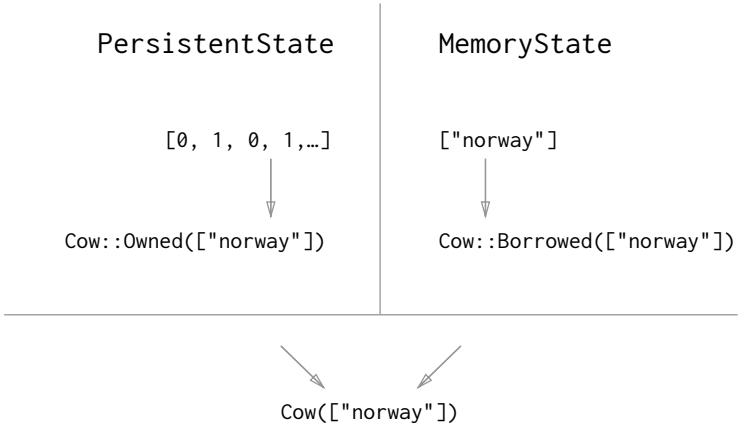

Figure 5.2: Rows read from `PersistentState` are deserialized from their on-disk byte representation, and returned in a form that transfers the ownership of the value to the caller. Values in memory are not serialized, and `MemoryState` instead returns pointers, removing the need to duplicate data. The `Cow` Rust structure represents either an `Owned` value, as in the first case, or a `Borrowed` value—a pointer.

What if, on the other hand, a row does not exist in memory to begin with? This would be the case when data is retrieved from durable storage: after reading a row in its on-disk representation and de-serializing it to a value in Soup's data format, where is that very value stored? The *data* used to create the value exists on disk, but the actual memory representation of the value was just created. While `MemoryState::lookup` would want to return a reference to an internally stored value, `PersistentState::lookup` would rather want to hand over ownership of the value to the caller. Enter the `Cow`[4] —a *clone-on-write* pointer from Rust's standard library that helps with this exact purpose, by allowing data to be represented either as `Borrowed` in the case of `MemoryState` or `Owned` for `PersistentState`.

Additionally, the rows returned from `State::lookup` are wrapped in a `LookupResult` enum, representing either a found or a missing value. The latter is never relevant for `PersistentState`, which is only used to represent fully materialized state. Note that a *miss* does not signify that the value does not exist, it simply means that this particular `State` does not have it, while a `State` instance further up the data-flow graph does.

[4] https://doc.rust-lang.org/std/borrow/enum.Cow.html

```
// Before:
pub enum LookupResult<'a> {
    Some(&'a [Rc<Vec<DataType>>]),
    Missing,
}

// After:
pub enum RecordResult<'a> {
    Borrowed(&'a [Rc<Vec<DataType>>]),
    Owned(Vec<Vec<DataType>>),
}

pub enum LookupResult<'a> {
    Some(RecordResult<'a>),
    Missing,
}
```

Listing 5.4: Prior to the introduction of `PersistentState`, reads from `State` would always result in a borrowed, reference counted value. With `PersistentState` the caller is instead responsible for retaining ownership of the value.

Using `LookupResult`

Introducing the potential of a returned row being either `Borrowed` or `Owned` has its downsides. For one, callers would have to handle both branches, as the *type* contained in a borrowed value is different from the *type* in an owned one. In the former, callers would always have to clone the value to hand it over to someone else, whereas in the latter, they could simply *move* it out. The difference here comes from the expectations of the value: a borrowed value implies that someone wants to retain ownership of it—*e.g.*, it has to be cloned to give ownership to someone else—while an owned value is the responsibility of the caller.

```
if rows.len() > 0 {
  match rows {
    RecordResult::Owned(mut rows) => {
      out.push(Record::Negative(rows.swap_remove(0)))
    }
    RecordResult::Borrowed(rows) => {
      out.push(Record::Negative((*rows[0]).clone()))
    }
  }
}
```

Listing 5.5: With `RecordResult`, returned values can be either `Borrowed` or `Owned`, making callers responsible for handling both use cases.

As is often the case, this can be simplified by looking at what the callers are actually using the returned values for. The unpacked `rows` variable in listing 5.5 is—as the name implies—a list of rows. Taking this collection, and *iterating* over either all of, or a subset of the rows, before returning a new iterator over the potentially modified values, was by far the most common operation. Instead of delegating the responsibility for doing so to the callers, this can be implemented on `RecordResult` itself, greatly simplifying use cases like listing 5.5.

```
if let Some(row) = rows.into_iter().next() {
    out.push(Record::Negative(row.into_owned()));
}
```

Listing 5.6: The option of turning a `RecordResult` into an iterator is used to simplify the logic from listing 5.5.

5.4 Persistent state with SQLite

The first iteration towards a durable state implementation uses SQLite as its storage engine. Described in section 2.2, SQLite is a well-tested, heavily used system with a track record in everything from applications to other databases. SQLite uses B+trees internally—a reasonable, no frills data structure with easy-to-reason about performance guarantees. This makes it useful for an initial prototype, and will help us answering the question of whether a B+tree based `PersistentState` is feasible following the requirements defined in 5.2. We will use the Rusqlite [35] library for calling into SQLite from Rust.

5.4.1 Schema

SQLite is an embedded database, and requires no inter-process communication to function. When persisting data, SQLite writes directly to durable files. This requires exclusive locks to be held while writing (see section 2.2.2), eliminating the possibility of simultaneous updates from parallel locations to the same database. This is not an issue for Soup and `PersistentState`. Soup's `State` instances are completely standalone, and each `PersistentState` instance can operate against a separate SQLite database, avoiding the need for locks altogether.

SQLite—like SQL databases in general—require a strict schema to be defined at all times. How this schema looks is usually a result of what kind of queries a database needs to respond to, which in Soup's case depends on the indices a specific `State` instance has been given responsibility for. Each column in a Soup index will be given a column in the SQLite table, allowing flexible read queries on any of the given indices. The row itself will be stored in a separate column, serialized using `bincode` (see section 2.5).

5.4.2 Adding indices

Each call to `State::add_key` sets up that specific `State` instance for queries on the given set of columns. This primarily involves extending our SQLite schema with the newly given columns, while creating an actual SQLite index on the *column combination* itself. The latter is not strictly necessary: SQLite is able to retrieve data for any columns, regardless of existing indices. The performance would be exceedingly poor however, as it would require scanning the entire table.

As an example, consider a Soup base node with the three columns `(a, b, c)`. Queries further down the graph dictate that the base node needs to be able to efficiently read rows by the columns `(a, b)`, and by only `c`. After a series of inserts, our SQLite table for the base node's `PersistentState` might look something like table 5.1.

a	b	c	row
1	cat	norway	`bincode(1, cat, norway)`
2	dog	sweden	`bincode(2, dog, sweden)`
3	fish	denmark	`bincode(3, fish, denmark)`

Table 5.1: An underlying SQLite table in `PersistentStore` after a few inserts. `bincode()` is used to signify that the value is serialized in the bincode binary serialization format.

5.4.3 Operations

This section describes how our `PersistentState` implementation translates its `State`-interface methods into operations on top of SQLite.

Retrieval

After the hard work of building the indices has completed, a myriad of rows is only a `SELECT`-statement away. `PersistentState::lookup` takes a set of columns and values for those columns as arguments, which are then translated into a `SELECT`-query. Considering the example in our previous section, the key `(1, cat)` for the columns `(a, b)` would result in a query on the form of `SELECT row FROM store WHERE index_0 = 1 AND index_1 = "cat"`, which SQLite can then complete in a timely manner due to the index on `(a, b)`.

Insertion

Given a vector of values, `PersistentState` has to first extract the columns necessary for its current set of indices, so that their values can be translated to SQLite friendly types. These can then be used to build an `INSERT`-query on the form of `INSERT INTO store (index_0, index_1, row) VALUES (...)`, where `row` is a binary representation of the entire vector, serialized using `bincode`.

Removal

Similar to lookups, removals need to first build a `WHERE`-clause by extracting the index column values from the target row, which can then be used to perform a `DELETE`-statement on the form of `DELETE FROM store WHERE index_0 = 1 AND index_1 = "cat"`.

Processing insertions and removals

All mutations need to be done within a transaction in SQLite, and operations performed without an explicit transaction are implicitly given one. Performing a single transaction is potentially expensive in SQLite given its strong durability guarantees, where each transaction will incur a `fsync` operation to make sure updates are successfully persisted before returning. Instead, `PersistentState` batches all mutations required for a single packet (per base node) into one transaction. This is done using the `State::process_records` method described in 5.3.1, as shown in listing 5.7.

```
fn process_records(&mut self, records: &Records) {
    let transaction = self.connection.transaction().unwrap();
    for r in records.iter() {
        match *r {
            Record::Positive(ref r) => {
                Self::insert(r.clone(), &self.indices, &transaction);
            }
            Record::Negative(ref r) => {
                Self::remove(r, &self.indices, &transaction);
            }
        }
    }

    transaction.commit().unwrap();
}
```

Listing 5.7: Multiple insertions and removals are wrapped in a transaction.

Counting rows

An accurate count of the total rows in the database can be retrieved using an SQL `COUNT`-query on the form of `SELECT COUNT(row) FROM store`.

5.4.4 Replacing the Soup write-ahead log

Soup already writes all updates to durable storage in the form of a write-ahead log. After an unexpected failure, Soup replays entries in the log to recover its state to what it was prior to crashing. This is far from optimal for long running applications (which most databases are), where recovery time would simply continue to grow unbounded. Relying on SQLite for durability would let Soup recover directly from SQLite's database files—a much faster operation than replaying the entire log.

Modern SQLite versions make use of a write-ahead log to ensure durability (see section 2.2.3) while maintaining high write performance. Relying on SQLite's WAL instead of Soup's requires some refactoring however: up until now Soup has sent out acknowledgments of writes as soon as the write is merged into a batch by the group commit protocol, prior to inserting the packet itself into Soup's data flow graph. Materialization into `PersistentState` happens after later, while the packet is being processed by a base node. This comes with a minor write latency penalty regardless of the `State` implementation, as more work needs to happen before an acknowledgment can be sent.

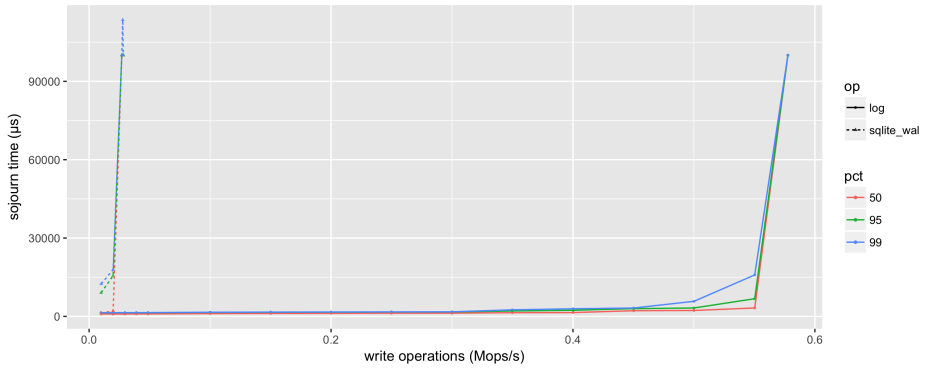

Figure 5.3: Write-only throughput measured using the `vote` benchmark. `log` relies on Soup's WAL for durability and stores base node state in-memory. `sqlite_wal` uses SQLite for durability, storing all base node state on persistent storage.

Intuitively, one might expect writes to SQLite to be at least somewhat slower than writes to Soup's regular write-ahead log, simply from the fact that the latter needs to write less data to disk. At the same time, the WAL implementation in SQLite is surely far more sophisticated than Soup's. SQLite's WAL consists of a series of frames, allocated ahead of time to avoid unnecessary resizing at every insertion. Soup's WAL is on the other hand purely an ever-growing sequential file, where entries are appended to the end of the file for each new packet. So why—as shown in figure 5.3—is SQLite so much slower?

Checkpointing

With an arsenal of profiling tools at our disposal, guessing is unnecessary. A flame graph built from profiling data recorded with `perf` quickly highlights the two main culprits: checkpoints and B-tree updates. As mentioned in section 2.2.3, SQLite automatically transfers data from the WAL to its main database file once the WAL exceeds a certain threshold. Checkpointing—similar to everything else in SQLite—is a synchronous operation, incurring a significant latency penalty once it happens. Regardless, checkpointing is a necessary evil. Up until the point of a checkpoint, reads have to refer to both the content in the WAL and the content in the main database. Delaying the automatic checkpoint operation does not make much of a difference either, as it results in more content to copy over when the checkpoint finally occurs.

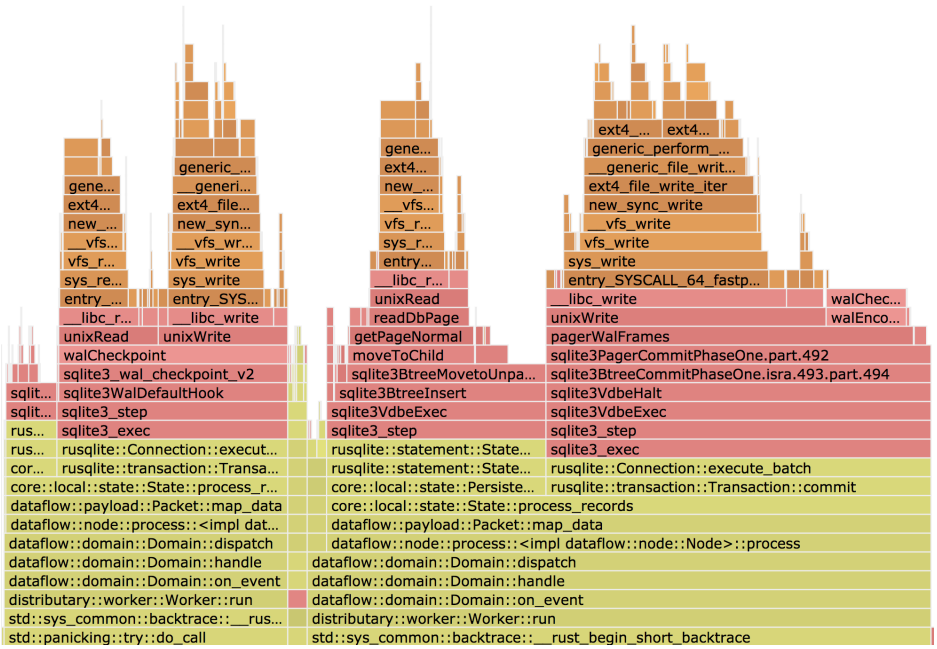

Figure 5.4: A flame graph highlighting time consuming functions in `PersistentState`.

Does checkpoints have to happen synchronously, before a write acknowledgment is sent to a client? In reality, they do not: taking a checkpoint does not make any difference in terms of durability. Instead, `PersistentState` can issue a checkpoint manually after a certain amount of time, but *after* acknowledging any outstanding writes. This had a minor effect on throughput, taking it from a measly 27k ops/s, to at least 42k ops/s—still far too slow. The checkpoints still happen synchronously, pausing processing at that specific `PersistentState` instance for a significant amount of time.

That leads to the question of whether checkpoints have to happen in the same thread as regular processing altogether. The SQLite manual mentions the possibility of taking checkpoints in a separate thread, by incurring manual checkpoints using the `sqlite3_wal_checkpoint_v2` method. While this sounds promising, it only helps if regular processing can continue while the checkpoint is being taken. SQLite describes four different checkpoint modes[5]:

- `SQLITE_CHECKPOINT_PASSIVE`: Checkpoints as many frames as possible without taking any locks.
- `SQLITE_CHECKPOINT_FULL`: Waits until any current database operations are finished, then holds an exclusive write-lock while checkpointing.
- `SQLITE_CHECKPOINT_RESTART`: Similar to the previous mode, but waits with

[5] https://www.sqlite.org/c3ref/wal_checkpoint_v2.html

finishing the checkpoint until all readers are accessing the main database file—and not the WAL—forcing any new writers to restart the WAL.

- `SQLITE_CHECKPOINT_TRUNCATE`: Same as the `RESTART` mode, except it also truncates the log file before restarting.

While the last three checkpoint modes hold write locks throughout the duration of the checkpoint, effectively resulting in the same operation as a synchronous checkpoint, the `PASSIVE` mode does not. It also has the possibility of not checkpointing any frames whatsoever either though, which is not very helpful. This makes it heavily workload dependent, as it requires a low enough throughput to allow "breaks" in the write processing where a checkpoint can happen. Without that being the case the WAL will simply continue to grow.

Updating indices

The other issue highlighted in the flame graph in figure 5.4 is index updates. While writing to a write-ahead log is purely sequential, updating B-trees is not. This is a significant difference from Soup's write-ahead log, where writes to the log only require sequential writes, instead of the random reads **and** writes caused by inserting into a B-tree. Even if maintaining SQLite's B-tree indices could be postponed to the checkpoint stage (which would likely degrade read performance), this would simply lead to a longer checkpoint, which, as the previous section points out, still results in pauses in regular write processing.

5.4.5 Relaxing SQLite's durability guarantees

To ensure durability, SQLite waits until writes have been fully persisted to durable storage before returning. Depending on the underlying storage medium, this might come with a quite hefty latency penalty, as the previous sections have shown. Let us now move to the other end of the spectrum, and investigate how SQLite fares with minimal durability guarantees. Instead, we will rely on Soup's regular write-ahead log for durability, and only use SQLite to avoid having to store base node state in memory.

Synchronization

Whereas the previous experiments ran with SQLite's `synchronous` option set to `FULL`, which ensures that all writes are safely persisted before returning, we will now make use of `synchronous = OFF` instead, which should significantly decrease the latency of writing to SQLite.

Foregoing atomicity

SQLite provides two main options to ensure atomicity: a rollback journal (section 2.2.2) and a write-ahead log (section 2.2.3). Whereas the previous section made use of the latter, we will now try a third option: no journal at all. Similar to `synchronous = OFF`, this is far from safe in the event of crashing, but is altogether a more useful comparison while relying on Soup's WAL for durability. If this is still too slow, then chances are it is going to be hard to achieve our requirements using SQLite no matter what.

Intermediary results

Whereas our SQLite WAL experiment only reached about 40k writes/s, the current setup is at least able to push past 110k writes/s. While an improvement, this is still far slower than Soup's regular write-ahead log.

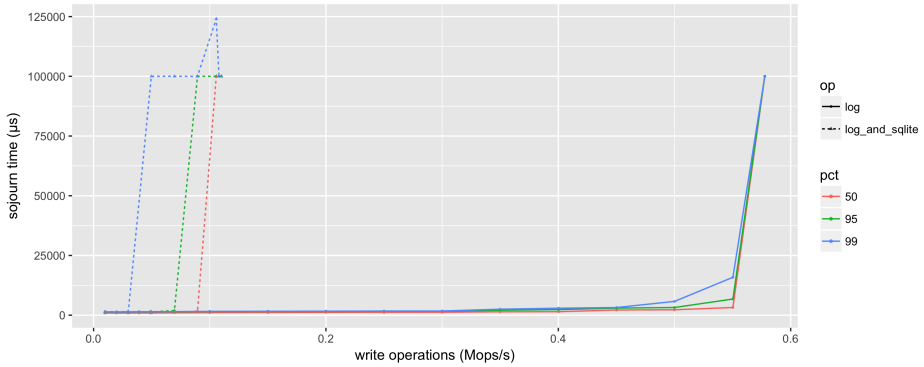

Figure 5.5: Write-only throughput measured using the `vote` benchmark, comparing in-memory base nodes to persistent base nodes using SQLite—both examples use Soup's WAL for durability.

Closing remarks

The fact that SQLite works well out of the box without a lot of tuning is definitely one of its strengths. Regardless, there are a few options that can be tweaked, both ahead of compilation and at runtime. While disabling SQLite features not needed for `PersistentState` and removing all mutex code had a slight positive impact, the improvements were far from significant enough to make a solid dent in the total throughput. Maintaining SQLite indices—and thus randomly reading from and writing to disk—on the main path is simply too slow. Does this imply that directly maintaining B-tree on-disk index structures built specifically for Soup would perform poorly as well? Possibly. While SQLite does have a lot of overhead needed to support a wide range of features,

and an even wider array of systems, profiling clearly shows that the bulk of processing time is spent interacting with durable storage as a result of B-tree maintenance.

5.5 Persistent state with RocksDB

How do we improve upon SQLite's low write throughput? One option would be to only write to durable storage from background threads, avoiding the hit in latency that comes from writing to disk. Instead, writes to `PersistentState` could fill up an in-memory queue structure, which would then later be flushed to persistent storage by background threads. Read operations could then retrieve data from both the in-memory queue and the slower—but persistent—SQLite. This might sound familiar, as it is effectively a naive sketch of the log-structured merge-tree data structure described in section 2.3, where writes are initially kept in an in-memory buffer and later flushed to disk by background workers. That leaves us with a good next step: embedding a storage engine that makes use of LSM-trees internally, RocksDB.

While RocksDB and its LSM-trees should improve `PersistentState`'s write throughput, it could at the same time introduce a further penalty to read latency. This might not be too problematic though, as reads from `PersistentState` should be more of a rare occurrence than writes: every update Soup sees needs to be persisted to durable storage, whereas only a subset of reads need to access `PersistentState` at all—most results should be served by partial nodes further down the data-flow graph.

5.5.1 Secondary index scheme

Whereas SQLite includes support for a large subset of SQL features, RocksDB's API is on a lower abstraction level. Limited to the functions necessary to maintain a sorted key-value store, implementing support for secondary indices is more of a challenge than with SQLite, where support for both primary and secondary indices is built in. Building higher level abstractions on top of key-value stores is becoming more and more common however, with plenty of inspiration to be found in existing projects, as described in section 3.1. Both CockroachDB [14] and MyRocks [31] implement SQL-like functionality on top of RocksDB, with sturdy indexing schemes at the core.

Prior to describing the final implementation, let us first iterate our way towards a working index scheme. The next few sections describe separate, working, implementations, where each step improves upon the last.

Collection of rows

Representing unique indices using a key-value store is fairly straight forward, as each key maps to a single value. With secondary indices, this is no longer the case, and a single key would need to represent multiple values. With key-value stores where

subsequent insertions are still retrievable with a single read operation this would require no further abstractions. In RocksDB however, each insertion effectively overwrites any existing values that might exist for that key. So how do we emulate an API mapping a single key to multiple values with the tools we have at hand? For a start, we could do exactly that, and store multiple values in an array below each key. Insertions would then retrieve the array of existing values, append the given value to that array, and write it back to the same key—as shown in listing 5.8.

```
Lookup(key):
  raw_values = Get(key)
  deserialize(raw_values) or []

Insert(key, value):
  # Retrieve and deserialize existing values:
  raw_values = Get(key)
  values = deserialize(raw_values) or []

  # Append our new value, serialize the collection and put it back:
  values += value
  Put(key, serialize(values))
```

Listing 5.8: A naive secondary index implementation, where each key contains a collection of values.

Unfortunately, this comes with the same write performance penalty as SQLite's B-tree index structures: each write now has to first perform a random read *before* the new value can be written. This completely removes any performance benefits we might gain from RocksDB's write-friendly LSM-tree structure.

Merge operator

In addition to the more common `Get` and `Put` operations, RocksDB supports an atomic read-modify-write operation through the `Merge` method[6]. This leads to the same result as in our previous implementation—with each key mapping to a collection of values—without any random reads while writing. Instead, RocksDB considers calls on the form `Merge(key, value)` as an *intent* to merge the values, and persists it as such. These intents can then be processed later, during background compactions (see section 2.3.5) and read operations.

[6] https://github.com/facebook/rocksdb/wiki/Merge-Operator

```
Lookup(key):
  raw_values = Get(key)
  deserialize(raw_values) or []

MergeOperator(key, existing, operations):
  # Retrieve and deserialize existing values:
  values = deserialize(existing) or []

  # Then, append all Merge(key, value) operations:
  for op in operations
    values += deserialize(op)

  # Finally, serialize and return the new list:
  serialize(values)
```

Listing 5.9: A RocksDB merge operator, appending any given values to an array.

This moves some of the work from writing to reading, as a read operation now has to find all merge operands and process them before returning. Unfortunately, this also introduces an unnecessary amount of serialization. RocksDB does not know how to deal with anything but byte streams, and so both the input and output to the merge operator have to be serialized, regardless of the fact that the final result will have to be deserialized again when read by the caller. This could potentially be improved through a smarter serialization scheme, where values could be appended to the serialized array without deserialization, but let us first consider an alternative without the merge operator altogether.

Iteration

Up until now we have considered options where each index value takes up a single key, through various methods of making the value portion of a key-value pair plural. This has shown to be quite sub-par, and it would certainly be an improvement if we could avoid storing a collection below each key. This is possible through *iteration*. By varying keys slightly to ensure uniqueness, while still making sure keys corresponding to the same index value are placed next to each other, we can *seek* to the first key for a specific index value, and continue to iterate through its subsequent siblings until we reach a separate key altogether. Retaining uniqueness among keys is not completely straight forward, and while we will consider this in further detail in section 5.5.3, we we will for now make do with a monotonically increasing sequence number, where each subsequent insertion increments the sequence number and appends it to the original key.

Consider an example table with three columns and two indices: `(id, name, country)`, with a primary index on `id` and a secondary index on `country`. Table 5.2 shows how

the structured schema is converted to a flattened key-value space by inserting additional rows for each index. Notice that the keys for the primary index do not need any additional suffixes to ensure uniqueness—its keys are already unique.

id	name	country
1	ola	norway
2	kari	norway

key	value
1	(1, ola, norway)
2	(2, kari, norway)
norway-1	(1, ola, norway)
norway-2	(2, kari, norway)

Table 5.2: Flattening a structured table with multiple index into a key-value scheme.

```
Lookup(key):
  values = []
  iterator = Iterator()
  iterator.seek(key)
  while iterator.valid():
    (next_key, next_value) = deserialize(iterator.next())
    if not next_key.starts_with(key):
      break

    values += next_value

  values

Insert(key, value):
  seq++
  new_key = serialize(key + seq)
  Put(new_key, serialize(value))
```

Listing 5.10: Implementing an indexing scheme using iteration.

5.5.2 Prefix iteration

While iteration is a promising concept, the performance guarantees it provides are potentially sub-par. Consider how an LSM-tree organizes its data: whereas each SS-table is sorted, separate files often overlap. This implies that an iterator has to consider every available file in its tree while iterating. To improve upon this, we can provide RocksDB with a way of partitioning out the areas that we want to iterate within, through use of *prefix extractors*. In our case, these partitions are separate index keys, such as `norway` in the example shown in table 5.2.

With a defined way of extracting prefixes, RocksDB is free to organize data how it sees fit to optimize for prefix iteration. While iteration usually guarantees a total order, this guarantee is restricted to iteration within the same key when a prefix extractor is given. Bloom filters (see section 2.3.6) can then be defined on prefixes instead of individual keys, greatly reducing the amount of SS-tables that have to be considered when iterating. Additionally, developers can choose to replace RocksDB's fundamental LSM-tree structures with data structures optimized for prefix iteration, such as hash tables where each prefix is a key.

Extracting a prefix

What signifies a prefix varies wildly between different applications. Instead of attempting a generalization, RocksDB leaves the extraction up to developers, through what is referred to as a *slice transform*: a function on the form of $S \rightarrow S'$, where S is a key and S' is a potentially modified version of S. For `PersistentState`, S' corresponds to the original key, with any appended suffixes stripped away, as shown in table 5.3.

key	prefix	value
1	1	(1, ola, norway)
2	1	(2, kari, norway)
norway-1	norway	(1, ola, norway)
norway-2	norway	(2, kari, norway)

Table 5.3: The extracted prefixes strip away the previously appended key suffixes.

This gives us the ability to iterate purely within a prefix. Instead of having to manually check whether our next iteration step reaches a sibling key, we can now seek directly to the prefix and RocksDB will ensure that all given keys stays within that prefix. At the same time, we do not have to take care to make sure that the first key—*e.g.*, `norway-1`—is ordered before any other keys: RocksDB ensures that our prefix iterator steps through all keys within that prefix. To define a correct prefix extractor, RocksDB defines four boolean properties that need to hold:

1. $key.starts\_with(prefix(key))$
2. $Compare(prefix(key), key) \leq 0$
3. $Compare(k1, k2) \leq 0 \implies Compare(prefix(k1), prefix(k2)) < 0$
4. $prefix(prefix(key)) \equiv prefix(key)$

Every key has to go through a defined slice transform at some point, so an efficient implementation is crucial. This is not completely trivial though, as both the prefix

transform's input and output are byte streams. How do we separate out the index part of a key from a byte stream? A naive way to do so would be to first deserialize the key, extract the index values, before finally serializing and returning the prefix in byte form (shown in listing 5.11).

```
PrefixTransform(raw_key):
  try:
    key, suffix = deserialize(raw_key)
  catch:
    # raw_key might already have gone through
    # a prefix transform (property 4):
    return raw_key

  serialize(key)
```

Listing 5.11: A naive implementation of a prefix transform.

Deserializing the entire key just to strip off the suffix requires an unnecessary amount of allocations. In the current example the suffix is an integer sequence number, which we always know the byte size off, giving us the alternative of slicing away the last X bytes from the key to build a prefix, where X is the size of our integer suffix. This would help maintain the three first prefix transformation properties, while failing the last: how do we ensure that we only slice away the last bytes once when performing $prefix(prefix(key))$? Additionally, as we will see in later sections, there are cases where we would like the suffix to have a variable byte size.

This leaves us with a requirement for a more generic solution, which we will solve by slightly worsening the space complexity of our key scheme, through the introduction of a *size* segment. Consider a 64-bit integer primary key, which bincode (see section 2.5) uses 12 bytes to serialize. By prefixing the key with the serialized size, 12, the slice transform knows how many bytes correspond to the index value of the key, which it can then use to remove any suffix values.

```
PrefixTransform(raw_key):
  # bincode uses 8 bytes to encode
  # the size value itself, a 64-bit integer:
  size_offset = 8
  key_size = deserialize(raw_key[..size_offset])
  prefix_length = size_offset + key_size
  # Finally, strip away the suffix:
  raw_key[..prefix_length]
```

Listing 5.12: With the byte size of the relevant portion of the key included, `PrefixTransform` only needs to deserialize a single integer to be able to slice away the suffix.

5.5.3 Separating indices

The index scheme described so far has glossed over the potential for conflicts between indices covering the same amount of columns, with the same types. Consider a table with two integer columns, `(a, b)`, and an index on each column. How do you separate values pointed to by the index on column `a` from values pointed to by the index on `b`? In the example shown in table 5.4 this would lead to a conflict: both index `a` and index `b` have at least one key with the same value.

a	b
1	2
2	2

Table 5.4: Two indices with the same number and type of columns would collide under a flat key-value space.

Both CockroachDB [16] and MyRocks [32] solve this by including a unique index ID in the key (shown in table 5.5). This is an elegant solution with low overhead—a small integer value would be enough to cover a large amount of indices. Another option would be to put each index in a separate column family—the RocksDB equivalent of a table. Whereas a secondary index might have a large amount of values prefix, a unique index can have only one. With the option of opening each column family with different options, indices can be tuned differently based on similar patterns. Keeping a separate column family per index also simplifies the rare case where all rows in the system have to be iterated through, as all the values can be retrieved from the first index, without having to filter out the pointers kept by other indices.

key	value
0-1	(1, 2)
0-2	(2, 2)
1-2	(1, 2)
1-2-…	(2, 2)

Table 5.5: Keys in CockroachDB and MyRocks are prefixed with unique index IDs to prevent collisions.

5.5.4 Ensuring unique keys for secondary indices

Up until now the secondary index keys have been suffixed with a monotonically increasing sequence number. While this works well as an example, it has a few issues. For one, how does `PersistentState` know what sequence number to start at after recovering? Recording the latest sequence number after each write would significantly degrade write throughput, while figuring out the last sequence number used upon recovering would increase the time it takes to recover.

Instead, `PersistentState` includes an *epoch* with the sequence number, which it increments and persists upon recovering. This ensures that the sequence number can start counting from zero after recovery, without any chance of collision. In short, the pair `(epoch, sequence number)` is always unique for a single index.

Maintaining a `(epoch, sequence number)` pair for each index in every case is also unnecessary. As long as keys are unique for the initial index—the primary index—the secondary indices can make use of the primary key as a suffix to ensure uniqueness. This is shown in table 5.6, where the secondary index on country includes the primary key as a suffix, to avoid collision. This also greatly speeds up the speed of removals: since secondary index keys can now be derived using the row and the primary key value, no unnecessary retrievals are needed to remove rows.

key	**value**
10	(10, bob, norway)
20	(20, anne, norway)
norway-10	(10, bob, norway)
norway-20	(20, anne, norway)

Table 5.6: The primary key, ID, is used as a suffix for the secondary index on country to ensure that separate keys are unique.

This is also how CockroachDB [16] and MyRocks [32] handle secondary index keys. What happens when the primary index does not have unique keys? While CockroachDB and MyRocks recommend against it, it is supported by using an automatically generated ID. This is where our previously mentioned `(epoch, sequence number)` pair comes in, by functioning as an automatically generated suffix to ensure uniqueness among keys in the initial index. With the exception of the extra storage required for the keys, this is completely free: unlike an incremented identification number, no extra writes or reads are incurred for our sequence number, as it is reset when recovering.

key	value
bob-0-0	(bob, norway)
anne-0-1	(anne, norway)
norway-bob-0-0	(bob, norway)
norway-anne-0-1	(anne, norway)

Table 5.7: In this example the table has no unique primary key, and instead uses the potentially duplicate `name` column as an initial index. Since `name` is not guaranteed to be unique, it is suffixed by a `(epoch, sequence number)` suffix. Secondary index keys are still suffixed with the entire key from the initial index to ensure uniqueness.

```
Recover:
  epoch = ReadEpoch()
  epoch++
  PersistEpoch(epoch)
```

Listing 5.13: The epoch is retrieved, incremented, and persisted again when `PersistentState` recovers.

5.5.5 Following index pointers: space versus performance

Having covered the contents of the key portion of secondary index rows, let us now consider the value. In MyRocks, retrieval of secondary index values require a subsequent lookup of the primary key row, to avoid having to store the value multiple times. This opens for a completely empty value portion: the key already includes everything needed to retrieve the full row.

An additional read for every secondary index lookup has the potential to be quite costly for large indices. With `PersistentState`'s iteration scheme, rows covered by the same index are co-located, opening for efficient caching behavior when a single secondary index key points to a large amount of rows. This is less of an improvement if each of the retrievals result in a second, and completely random, read. The second lookup has no such property and the values might be located anywhere. Other database systems often utilize a covering index: when all the columns needed exist in either the key portion of the primary or secondary index, no further reads are required. This is not possible for Soup, where nodes further down the query graph always require the full row representation.

key	value
1	(1, ola, norway)
2	(2, kari, norway)
norway-1	NULL
norway-2	NULL

key	value
1	(1, ola, norway)
2	(2, kari, norway)
norway-1	(1, ola, norway)
norway-2	(2, kari, norway)

Table 5.8: The leftmost table stores nothing for secondary indices and require an additional lookup for each row. The rightmost includes the entire row for each index and does not.

Instead `PersistentState` trades write amplification and space usage for read performance, by storing the whole row for each index. Insertions now have to write the whole row for every index, increasing the total storage size, but with the great advantage of never having to read more than once to read from a secondary index.

5.5.6 Operations

This section describes how the various `PersistentState` methods are implemented on top of the RocksDB storage engine.

Retrieval

The way retrieval is performed varies based on the index type. For unique primary key indices, each key maps to a single row, where the row can be retrieved using a RocksDB `Get` operation. Other indices need to use a prefix iterator, as described in section 5.5.2, to ensure that all of the corresponding values are retrieved.

```
Lookup(columns, key):
  index = index_for(columns)
  prefix = serialize_prefix(key)
  if index.is_unique:
    raw_row = Get(prefix)
    deserialize(raw_row) or []
  else:
    prefix_iterator(prefix).map(value -> deserialize(value))
```

Listing 5.14: Retrieving one or more rows from `PersistentState` backed by RocksDB.

Insertion

The different parts of our key-value scheme comes together when inserting rows. A unique primary key is built up, with an incremented sequence number suffix included if its values are not unique by themselves. Then, a RocksDB `Put` operation is performed once for each index, with the same value: the serialized representation of the row.

```
Insert(row):
  # Extract the key portion of the row:
  primary_key = build_primary_key(row)
  if has_unique_index:
    # The first index is unique, so no sequence number or epoch needed:
    serialized_key = serialize_prefix(primary_key)
  else:
    # Otherwise, increment the sequence number and use that in the key:
    sequence++
    serialized_key = serialize_key(primary_key, epoch, sequence)

  serialized_row = serialize(row)
  # First write the primary key row:
  Put(serialized_key, serialized_row)

  # Then insert a row for each secondary index:
  for index in secondary_indices:
    key = build_key(row, index)
    serialized_key = serialize_key(key, serialized_key)
    Put(serialized_key, serialized_row)
```

Listing 5.15: Insertions write one row per index.

Removal

Removals undo the work done by an insertion operation, by cleaning up rows for each index maintained by a `PersistentState` instance. Note that a removal in Soup is given the entire row, and not a key, with the expected outcome that it will delete a *single* value. This means that we need to take care to only remove that exact value—and any references to it—for non-unique indices.

```
PerformRemove(row, primary_key):
  # Delete the primary index row first:
  Delete(primary_key)

  # Then delete any secondary index references:
  for index in secondary_indices:
    key = build_key(row, index)
    serialized_key = serialize_key(key, primary_key)
    Delete(serialized_key)

Remove(row):
  primary_key = build_primary_key(row)
  prefix = serialize_prefix(primary_key)
  if has_unique_index:
    PerformRemove(row, prefix)
  else:
    for (key, value) in prefix_iterator(prefix):
      if deserialize(value) == row:
        PerformRemove(row, key)
```

Listing 5.16: Removals clean up all the index references pointing to the given row, after removing the row itself.

Counting rows

RocksDB, being a key-value store, does not have an equivalent of the *COUNT* operator in SQLite. To retrieve an exact count of all rows, one would need to either maintain a separate variable (which would then need to be persisted potentially on every write), or iterate through the entire collection of rows—a potentially costly operation. RocksDB does on the other hand have an estimated internal property, `rocksdb.etimate-num-keys`, which can be retrieved trivially. At the moment, a total row count is only needed for debugging and statistical purposes in Soup, making an estimated count sufficient.

5.5.7 Replacing the Soup write-ahead log

Section 5.4.4 attempted to replace Soup's write-ahead log with SQLite's, ensuring insertions and removals were safely persisted to SQLite prior to sending out any write acknowledgments. This turned out to be too slow: in addition to writing records to the WAL, SQLite also had to update its B-tree indices before returning. This is not the case with RocksDB, where data is first written to an in-memory buffer and only later flushed to disk by background threads. While this greatly reduces the gap between `PersistentState` and regular Soup, RocksDB still has to potentially write N times more data to disk than Soup's regular WAL, where N is the amount of indices

`PersistentState` maintains.

To prevent synchronizing to durable storage more than necessary, the records from a single packet result in a single `WriteBatch` (see section 2.3.3)—persisted once towards the end of `PersistentState::process_records`.

5.5.8 Building new indices

Indices added after rows have already been inserted into `PersistentState` need to be *built*, to ensure that they start serving reads for existing rows right away. This is done by iterating through all of the existing rows while inserting index pointers into the newly created column family. Finally, meta information about the index itself is persisted to RocksDB *after* building the index, preventing recovery from trying to rebuild the index after a failure.

```
AddKey(columns):
  if columns in existing_indices:
    return

  index_id = length(existing_indices)
  column_family = create_column_family(index_id)

  if index_id > 0:
    for (primary_key, value) in all_rows():
      row = deserialize(value)
      key = build_key(row, columns)
      serialized_key = serialize_key(key, primary_key)
      Put(column_family, serialized_key, value)

  existing_indices += columns
  PersistMeta()
```

Listing 5.17: Adding a new index to `PersistentState` builds the index using any existing rows, before finally persisting the index to RocksDB.

It might be tempting to wrap the entire index construction in an atomic `WriteBatch` for multiple reasons:

- The index would either be completely built or not built at all.
- The WAL would only have to be synchronized once—when committing the `WriteBatch`.

Unfortunately—at least for this particular case—a `WriteBatch` keeps all updates in memory until it is committed. This raises a potential issue: an index could only be built if all existing rows fit in memory. On the other end of the spectrum, committing

every row to disk incrementally (as in listing 5.17) would be incredibly slow. Instead, we will reach for a compromise and create a new `WriteBatch` every N rows, where $N * RowSize < MemorySize$.

Separating the index building into multiple batches voids the guarantee of atomic index construction. Consider the case where `AddKey` crashes while trying to build a new index, after having created a new column family for it. If `PersistentState` continued to use this partially built index after recovering, its returned results would be faulty. This is where `PersistMeta()` comes in. By only writing metadata about the index *after* it has been built, we ensure that any future recovery processes know how to differentiate a partially built index from a complete one. During recovery we can then check whether the index count corresponds to the column family count, and if it does not, simply drop the entire column family and rebuild the index again later.

5.5.9 Background threads

With the exception of its write-ahead log, RocksDB first buffers writes to an in-memory MemTable before persisting data to durable storage. This comes with performance benefits, by avoiding the latency penalty that often comes from writing to disk. MemTables have to be flushed to durable storage at some point however, which is handled by *background threads*. These are also responsible for compacting on-disk SS-tables (see section 2.3.5).

RocksDB's background threads are shared across RocksDB instances within the same process. This is helpful for Soup, where a single Soup instance might contain a multitude of sharded base nodes, leading to a core-constrained system if all nodes were given a background thread each. RocksDB divides its background threads into two priority partitions, where `HIGH` is reserved for flushing memtables and `LOW` is assigned for compactions. The amount of threads in both is configurable, with different hardware and varying workloads requiring different thread numbers. Soup—like RocksDB—defaults to a single background thread per process, with a recommendation of increasing it left to its end-users.

Chapter 6

Recovery

This chapter describes the current recovery scheme used in Soup, followed by a look at how it improves when the base tables are stored on durable storage instead of in volatile main-memory. Finally, the rest of the chapter is reserved for snapshotting—a significant improvement to Soup's recovery capabilities.

6.1 Write-ahead log

Soup appends all updates to a durable write-ahead log prior to sending out write acknowledgments. Packets are buffered up using a group-commit scheme [21] to amortize the I/O cost of writing to durable storage. Commits result in a merged packet injected into the data-flow graph, ensuring that all changes are persisted to disk before being accessed by readers.

Merging the data from multiple packets before writing to the WAL drastically increases the throughput of the system. Instead of requiring a single write to durable storage per packet, Soup only needs to write once per batch. At the same time, it reduces the total amount of packets fed through the data-flow graph, allowing each node to operate on a larger batch of records at a time.

6.1.1 Log based recovery

In a traditional relational database management system, log recovery often involves multiple phases. Following a system like ARIES (described in section 3.2), log recovery is responsible for a wide variety of tasks, such as undoing aborted transactions. Log recovery in Soup is much simpler, with log entries only including the operations to be performed, and not the result they have on the database—a *command log* [54]. There are no `UNDO` nor `REDO` phases either: only committed transactions result in persisted log entries.

Log recovery is then a matter of replaying the writes from the persisted logs by feeding them into the data-flow graph's base nodes as if they were regular updates. Each line in the log corresponds to a packet that was once injected into the Soup instance's data-flow graph, now serialized using JSON.

```
[
  [{"Positive": [{"Int": 1}, {"Int": 0}]}],
  [{"Positive": [{"Int": 1}, {"Int": 1}]}],
  [{"Positive": [{"Int": 1}, {"Int": 2}]}]
]
```

Listing 6.1: An expanded line from one of the log files of a Soup application, corresponding to a single batched update with three records.

Log entries are separated by a newline character, where each line in the log is valid JSON. This lets the recovery process handle one line at a time in a buffered fashion, avoiding the need to store the entire log in memory. Similar to how the packets were created in the first place, individual updates from each log entry are batched into chunks, before finally being turned into separate packets that are passed on to the rest of the data-flow graph. This speeds up recovery performance by avoiding processing of a potentially large amount of small log entries.

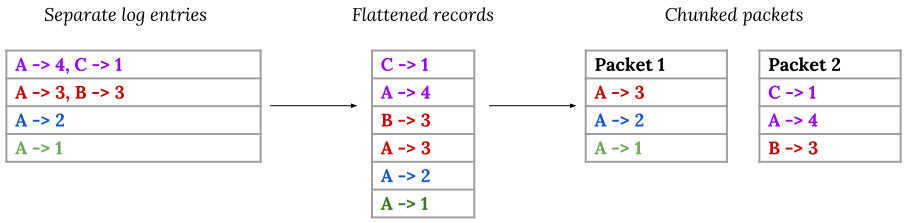

Figure 6.1: Log entries are flattened, then chunked into individual packets.

6.2 Persistent base nodes

RocksDB recovers content that resided in its in-memory MemTables at the time of a crash by replaying entries from its write-ahead log. This is a stark improvement from recovery using Soup's regular write-ahead log, where every entry from the beginning of time has to be replayed. Data that is already persisted to durable SS-tables at the time of a crash require no extra work—they can be read in the same manner after crashing as before.

What happens to Soup's partially materialized nodes? Similar to a regular caching solution, they start out completely empty after recovering. Subsequent requests gradually restore the nodes to a state resembling the one they were in before crashing. Fully materialized nodes require a complete view of the state at all times and need to be sent a full copy of the state when recovering.

Both this, and the fact that partially materialized nodes start out empty, lead to reduced initial performance—a target of improvement addressed in the next section.

6.3 Snapshotting

While `PersistentState` speeds up base node recovery, it comes with no improvement for recovery of nodes further down the graph. Partial nodes have to trigger a large amount of replays to restore their state early on, while fully materialized nodes need a complete copy of the state altogether before serving any reads at all.

`PersistentState` lets base nodes instantly recover to a recent point in time, capping recovery to the time it takes to go through recent updates. A similar solution for all materialized state would let nodes recover to a recent checkpoint, followed by reapplication of log entries to become fully up-to-date. In short, we need to be able to consistently *snapshot* the materialized nodes at any given point.

The implementation in the rest of this section is based on an earlier version of this thesis [25], submitted as a part of "TDT4501 Computer Science, Specialization Project".

6.3.1 Challenges

Main memory systems like VoltDB leverage checkpointing by persisting the transactional state of committed transactions, using log sequence numbers to be able to track which updates have been reflected on disk [54]. In Soup, state is materialized at a variety of nodes throughout the graph, and updates have no timestamps or sequence numbers attached to them. Updates propagate through the graph asynchronously, and a specific update is likely to reach different points in the graph at separate times. Taking a global snapshot of the entire graph simultaneously would mean capturing nodes at different logical points, as an update might be in the process of propagating throughout the graph at the time that the snapshot is initiated.

Soup's way of asynchronously propagating updates through its data-flow graph resembles the communication done in a distributed system. Being able to observe the global state in a distributed system—where access to a common clock is rare—is an immensely useful property, crucial to resolving a certain category of problems, such as deadlock detection and global checkpointing. Section 3.2.2 introduces a series of algorithms for snapshotting distributed systems, including the Chandy-Lamport [11] method of propagating an explicit snapshot marker throughout the system to ensure consistency, which we implement later in this chapter.

6.3.2 Algorithm

Taking a snapshot of a running Soup instance requires persisting the content of each materialized node in the current data-flow graph. This needs to happen at the same logical point in time across the graph—ensuring that every in-flight update is either propagated to *all* nodes in the graph—or none of them, leading to a consistent state after recovering from a failure. At the same time, taking a snapshot should not incur a too heavy performance cost on the running system and should definitely not stop the system from processing updates completely—for any period of time. This lets us derive a few base rules for our snapshotting algorithm:

1. Snapshots need to include exactly the same updates across the graph.
2. Snapshots should not significantly degrade the system's throughput.
3. Snapshots should complete in a reasonable amount of time.

We can then use these rules to build a snapshotting algorithm in incremental steps, starting from an example that fails to meet the defined criteria. Figure 6.2 shows an update propagating through the Soup data-flow graph. What would be the outcome if both of the partially materialized nodes—shown in a blue color—would snapshot their state at the exact moment shown in the graph? Whereas the leftmost domain has had time to process update `A`, the rightmost one has not. The two domains are at different *logical* points in time, and the snapshots would fail our first rule.

What if the system as a whole instead waited for the update to completely propagate through the graph before initiating the snapshot? This would successfully follow the

first rule, but fail the second: no new updates could be served until the snapshot has completed across the graph, halting the system's throughput.

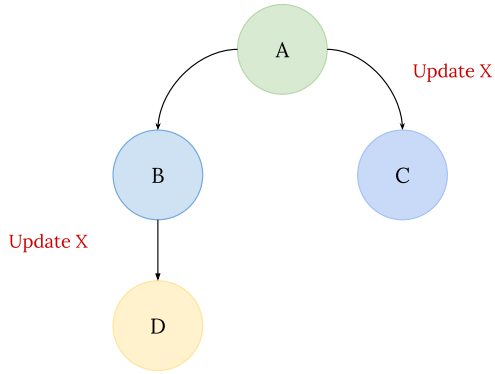

Figure 6.2: An update `X` propagates through the domains in the graph in an asynchronous manner. Domains *B* and *C* contain at least one materialized node and should be snapshotted.

Synchronous snapshotting

Soup's data-flow graph forwards updates over ordered FIFO channels, making it possible to rely on Chandy-Lamport's marker technique to determine which updates should be considered a part of a snapshot. Domains that receive the marker initiate the snapshot process right away, without any further processing of updates. This results in a global snapshot taken at the same *logical* point in time, even if the actual snapshots were instantiated at different *physical points*.

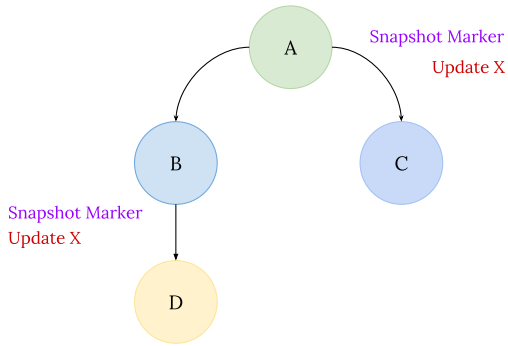

Figure 6.3: An update `X` is propagated through the data-flow graph, followed by a snapshot marker to ensure that domains with materialized nodes snapshot at the correct time.

The controller initiates a snapshot by issuing a `TakeSnapshot` marker to each of its base nodes, which is then propagated through the rest of the graph. After a snapshot completes, the controller persists its `snapshot_id` to durable storage, to ensure that it can inform domains which snapshot to recover from after a failure.

```
InitializeSnapshot:
  snapshot_id += 1
  for node in base_nodes:
    node.send(TakeSnapshot, snapshot_id)

  for node in base_nodes:
    node.wait_for_ack()

  persist(snapshot_id)
```

Listing 6.2: Initiating a snapshot from the controller.

Domains that receive a snapshot marker proceed with the snapshotting process immediately. After persisting the snapshot, nodes notify the controller that they have done so, letting it eventually discard log entries after confirming that all materialized nodes have successfully taken a snapshot of their state. Implemented naively, this would involve serializing each node's state at a domain and persisting these to disk—all while while blocking updates from the rest of the system (shown in listing 6.3).

```
TakeSnapshot:
  for node in nodes:
    if node.is_materialized:
      state = serialize(node.state)
      write(state)
      notify_controller(snapshot_id)
```

Listing 6.3: The beginning of a snapshot implementation for domains.

Asynchronous snapshotting

Our synchronous snapshotting algorithm fulfills the first snapshotting base rule through use of Chandy-Lamport's marker technique, by stopping further domain processing until the snapshot has completed. This involves writing the snapshot to durable storage—a potentially slow operation. To prevent this from significantly slowing down the system's throughput we would need to persist the snapshot in a separate computational unit, such as a thread, allowing the domain to continue its regular processing without pause.

The algorithm shown in listing 6.4 achieves this through a `SnapshotWorker` running in a separate thread, where the received state is serialized and persisted to durable storage

before the controller is notified of its completion.

```
SnapshotWorker:
  for event in listener:
      state = serialize(event.state)
      write(state)
      notify_controller(event.snapshot_id, event.state)

TakeSnapshotAsync:
  states = {}
  for node in nodes:
    if node.is_materialized:
      states[node] = node.state.clone()

  snapshot_worker.send(snapshot_id, states)
```

Listing 6.4: A `SnapshotWorker` serializes and persists snapshot in a thread separate from the regular domain processing.

Delayed snapshotting

Compared to the synchronous snapshotting algorithm—where processing is parallelized across all available domains—our asynchronous version restricts the number of parallel units to a fixed set of snapshotting workers. This number is likely to be far less than the number of domains, introducing a trade-off between snapshot completion time and the extra load induced on the system. The former is irrelevant as long as each snapshot completes before the next request arrives, and the focus should without doubt be on avoiding a potential performance hit to the processing throughput.

While the introduction of asynchronous snapshot workers move the bulk of the snapshot processing out of the domains' main thread, the state clone operation remains. This is a crucial part in maintaining the correctness of Chandy-Lamport's marker technique, and snapshots would not happen at the correct logical instant without it. Still, with snapshots happening roughly at the same physical time across the graph, the pause required from cloning a domain's entire state could have a significant impact on the system's total throughput.

Instead, we would like to amortize the performance penalty across a larger time range, by delaying the snapshotting process at each individual domain. While snapshots would still need to happen at the same *logical* point across the graph, the *physical* instant could vary. Naturally, this could be achieved by cloning the state immediately and only forwarding the result later on—without any gain at all. To actually spread out the cost of snapshotting we would need to delay the clone as well.

A clone of a domain's state has to be taken at some point, but preferably later than when the snapshot marker arrives. This would require the ability to travel back in time

from a state S_m to the original state when the marker arrived, S_n. In short, with $L_{n..m}$ signifying the updates that arrived from n to m, S_n can be re-created through $S_n - L_{n..m}$. The amount of work performed by a single domain would be higher, but in return the individual clone operations performed across all domains could be delayed, preventing a global performance penalty.

```
TakeSnapshotDelayed:
  states = {}
  for node in nodes:
    if node.is_materialized:
      current_state = node.state.clone()
      states[node] = current_state - processed_updates[node]

  snapshot_worker.send(snapshot_id, states)
```

Listing 6.5: A delayed implementation of `TakeSnapshotAsync` from listing 6.4. Updates arriving after the marker are stored in `processed_updates`.

Snapshot confirmations

The controller should be notified of all completed snapshots, as shown in listing 6.6. The snapshots are already persisted to disk at this point, removing the need to include data in the acknowledgment messages. The messages should on the other hand include the current snapshot identifier, so that the controller eventually knows when the entire graph has completed the same snapshot. At that point the snapshot identifier can be persisted to ZooKeeper, and any log entries for updates prior to the snapshot being taken can be discarded. Recovery is then a matter of first loading each materialized node's state from their local snapshot, followed by replaying the rest of the log entries available.

```
ReceiveSnapshotAck(domain_id, snapshot_id):
  snapshot_ids[domain_id] = snapshot_id
  if min(snapshot_ids) == snapshot_id:
    persist_snapshot_id(snapshot_id)
```

Listing 6.6: The controller listens for snapshot acknowledgments from snapshot workers, updating an internal data structure with a mapping from domain to `snapshot_id` on each received confirmation. When all domains have completed their snapshots, the controller persists the `snapshot_id`, so that it later on can be used for recovery.

Failure before discarding the log

In the event of a failure *after* the controller has persisted the snapshot ID to ZooKeeper, but *before* all nodes have managed to discard the correct log entries from durable storage, duplicate replaying of those log entries upon recovery is a possibility. To prevent this from happening, each log entry should include their domain's current snapshot ID, so that log entries corresponding to old snapshots can be ignored during recovery.

6.3.3 Implementation

The snapshotting implementation roughly follows along the lines of the asynchronous snapshotting algorithm described in section 6.3.2. Snapshots are initiated with a specific marker sent by the controller, domains process snapshots by cloning their nodes' state, and snapshot workers are responsible for serializing and persisting the snapshots. Finally, the loop is closed when the controller is notified of each domain's completed snapshots.

Initializing a snapshot

Snapshots are initialized by the controller using a special marker packet, `TakeSnapshot`. This happens at a regular interval, defined by the configuration option `snapshot_timeout`. The controller processes events in an event loop, and snapshots are triggered by emitting an event to this internal loop. This is done from a separate thread, which sleeps until it is time to take a snapshot.

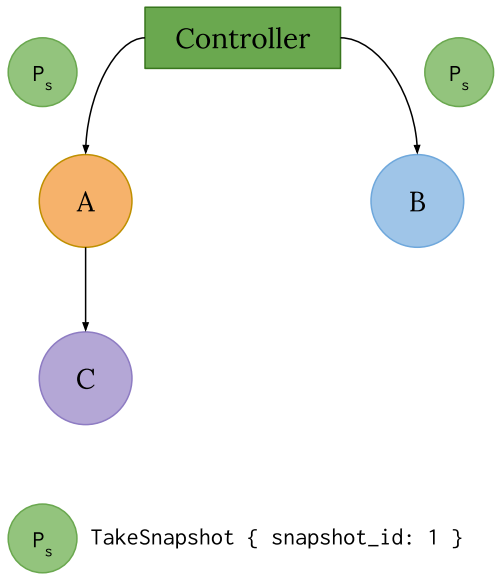

Figure 6.4: Snapshots are initialized when the controller sends a `TakeSnapshot` packet, which is forwarded through the data-flow graph by each domain.

When the controller's main loop receives a request to initialize a snapshot, it first makes sure that it has received all confirmations from previous snapshots before proceeding. Then it increments its `snapshot_id` and fires off a `TakeSnapshot` packet to each of the base node domains in its data-flow graph. No further blocking is required: snapshot acknowledgments are handled separately.

Domain snapshotting

Whenever a domain receives a `TakeSnapshot` packet, it immediately clones the state of each of its materialized nodes. Each domain then forwards the snapshotting packet to its children, ensuring that all materialized nodes eventually get snapshotted. Finally, it sends the cloned states to its `SnapshotWorker`, by issuing a `PersistSnapshotRequest`.

`TakeSnapshot` packets are forwarded to descendant domains through the egress nodes in the data-flow graph. Each domain is connected to its children through egress nodes and its ancestors through ingress nodes. With domains running in separate computational units, the snapshotting process can complete out of order at different domains.

Nodes and readers

The snapshotting algorithm describes persisting the state of materialized nodes. Although that is true, reality is slightly more nuanced. Materialized nodes in Soup can be

both internal and external, where the latter is represented by readers (see section 2.1.4). Both the internal nodes and the readers can be either partially or fully materialized, where only partial nodes require support from their parent nodes to fulfill certain queries.

Whether a node is partially or fully materialized does not make much of a difference for the snapshotting algorithm. On the other hand, internal and external nodes require slightly different snapshotting and recovery methods. This comes as a result of how state is stored within the two: whereas the former use Soup's own `State` data structure, the latter makes use of the `evmap` [37] library.

6.3.4 Performing snapshot requests

While each domain is responsible for gathering up the cloned copies of state needed to eventually recover from a failure, the actual serialization and persisting of snapshots happens in separate snapshot workers. The current implementation utilizes one snapshot worker per worker pool, resulting in one thread per running Soup instance. This could be increased as needed, but it is favorable that snapshots happen over a longer time to prevent reducing the system's throughput during the snapshotting process.

When a domain has finished cloning the state of its materialized nodes, it notifies its local snapshot worker through an asynchronous and unbounded buffered `channel`[1]. This ensures that snapshot processing can continue separately, without blocking the domain's regular workload. The snapshot workers receive and process `PersistSnapshotRequest` events one by one. The processing involves serializing the cloned state, persisting the serialized state to disk, and finally notifying the controller of a domain's snapshot completion.

6.3.5 Receiving snapshots confirmations

The controller listens for snapshot confirmations on a TCP socket normally used for coordination messages between the controller and its individual workers. Whenever it receives an acknowledgment packet from a snapshot worker, it stores the given `snapshot_id` in a `HashMap` mapping each domain containing at least one materialized node to their current snapshot ID. Note that domains can be sharded, so each instance in the map represents an individual shard of a single domain.

[1] https://doc.rust-lang.org/std/sync/mpsc/fn.channel.html

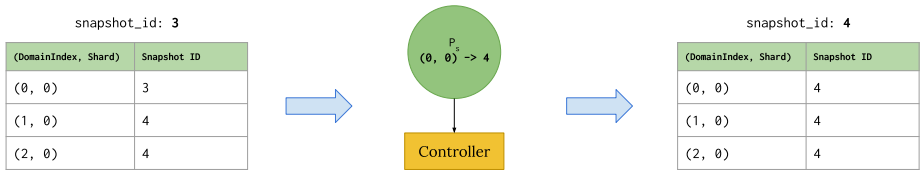

Figure 6.5: The controller keeps track of each domain shard's `snapshot_id` as the snapshot confirmations arrive. When all shards have been snapshotted, the controller's `snapshot_id` is persisted.

When all shards have completed its snapshot, the current `snapshot_id` is persisted to ZooKeeper, allowing the next snapshot to be initiated.

6.3.6 Logging and snapshotting

To be able to discern between log entries created before and after a given snapshot, log entries are annotated with the ID of the last snapshot persisted for that specific node. This lets the recovery process ignore any log entries created prior to the snapshot being restored.

```
[0, [[{"Positive": [{"Int": 1}, {"Int": 0}]}]],
[1, [[{"Positive": [{"Int": 1}, {"Int": 1}]}]]]
[1, [[{"Positive": [{"Int": 1}, {"Int": 2}]}]]]
```

Listing 6.7: Separate log lines for a given base node. Each line is on the format shown in listing 6.1, with the addition of a prefixed `snapshot_id`.

The snapshot ID used for log annotations is the ID of the latest snapshot initiated at that domain, regardless if the global snapshot with that ID has finished yet or not—it is *optimistic*. In the event of a failure, any unfinished snapshots are simply ignored, and the log entries are relied on instead. As an example, consider a domain that is in the process of taking a snapshot with ID X. A failure occurring after that domain has snapshotted, but before the entire graph has done so, would lead to each domain recovering the snapshot with ID $X - 1$, ignoring any log entries with $snapshot\_id < X - 1$.

6.3.7 Recovering from a snapshot

Snapshot recovery comes as a supplement to the log based recovery described in section 6.1.1. It is initialized when the controller sends a `StartRecovery` packet to each of its base node domains, containing the ID of the snapshot that was last persisted across the entire data-flow graph. The ID is read from ZooKeeper, ensuring that it survives across failures.

When the base node domains receive the recovery packets, they initialize the recovery process by first restoring snapshots, before replaying any updates that are left from log entries. Before this, the recovery packet is forwarded to the rest of the graph. Unlike log based replays, snapshots have to be restored at each materialized node across the *entire* graph—not solely at the base nodes. Forwarding the recovery packet prior to replaying log entries is crucial, as it ensures that snapshot recovery happens ahead of log based recovery for every node—not just the base nodes.

6.3.8 Serialization and deserialization of snapshots

Snapshots are serialized representations of each node's materialized state. The state itself is serialized from its Rust representation to a series of bytes using the `bincode` library, as described in section 2.5 and shown in listing 6.8.

```
// Implementing the Serialize and Deserialize traits makes it
// possible for bincode to serialize and deserialize the State struct:
#[derive(Clone, Serialize, Deserialize)]
pub struct State<T: Hash + Eq + Clone + 'static> {
    state: Vec<SingleState<T>>,
    by_tag: HashMap<Tag, usize>,
    rows: usize,
}

// Serialization:
let file = File::create(&filename)
  .expect(&format!("Failed creating snapshot file: {}", filename));
let mut writer = BufWriter::new(file);
bincode::serialize_into(&mut writer, &state, bincode::Infinite)
  .expect("bincode serialization of snapshot failed");

// And deserialization:
let file = File::open(&filename)
  .expect(&format!("Failed reading snapshot file: {}", filename));
let mut reader = BufReader::new(file);
bincode::deserialize_from(&mut reader, bincode::Infinite)
  .expect("bincode deserialization of snapshot failed")
```

Listing 6.8: State is serialized and deserialized using `bincode` [63].

6.3.9 Snapshot compression

Writing to and reading from disk is a significant part of the work being done during snapshotting and recovery. Both of these are naturally influenced by the performance

of the underlying storage medium. Compressing snapshots before they are persisted would lower the amount of bytes being written, at the expense of CPU cycles needed to compress the data before doing so.

Compression in Rust can be done using the `flate2` library[2], which supports a series of formats and backends. By working on streams, `flate2` composes well with the serialization library used for snapshotting, `bincode`.

```
// Serialization:
let file = File::create(&filename)
  .expect(&format!("Failed creating snapshot file: {}", filename));
let buffered = BufWriter::new(file);
let mut writer = ZlibEncoder::new(buffered, Compression::default());
bincode::serialize_into(&mut writer, &state, bincode::Infinite)
  .expect("bincode serialization of snapshot failed");

// And deserialization:
let file = File::open(&filename)
  .expect(&format!("Failed reading snapshot file: {}", filename));
let buffered = BufReader::new(file);
let mut reader = ZlibDecoder::new(buffered);
bincode::deserialize_from(&mut reader, bincode::Infinite)
  .expect("bincode deserialization of snapshot failed")
```

Listing 6.9: Serialization and deserialization of compressed snapshots using `bincode` and `flate2`.

6.3.10 Persisted data

Soup persists data to both local files and ZooKeeper as a part of the snapshotting and recovery process. Individual snapshots are written to durable storage locally at each domain, while the ID of the last completed global snapshot is stored in ZooKeeper, ensuring consensus between the replicated Soup controllers.

6.3.11 Diamonds in the data-flow graph

A single Soup node can receive multiple snapshot markers if at least two nodes have a common descendant in the data-flow graph. The snapshot implementation described so far immediately snapshots when a marker is received and ignores subsequently received snapshot markers. With packets propagating through the data-flow graph asynchronously, snapshotting after the first marker is received could produce inconsistent snapshots. Consider the case in figure 6.6, where the snapshot marker from node

[2] https://docs.rs/flate2/

B arrives at D ahead of the marker from C. Snapshots performed when the marker from B arrives would not include the propagated update from C, while delaying the snapshotting until the marker has arrived from C would potentially include update U_Y in the snapshot, breaking the properties defined by the Chandy-Lamport algorithm.

Instead, D should wait with snapshotting until it has received the marker from both of its parents, while blocking further updates from parents that it has already received the marker from.

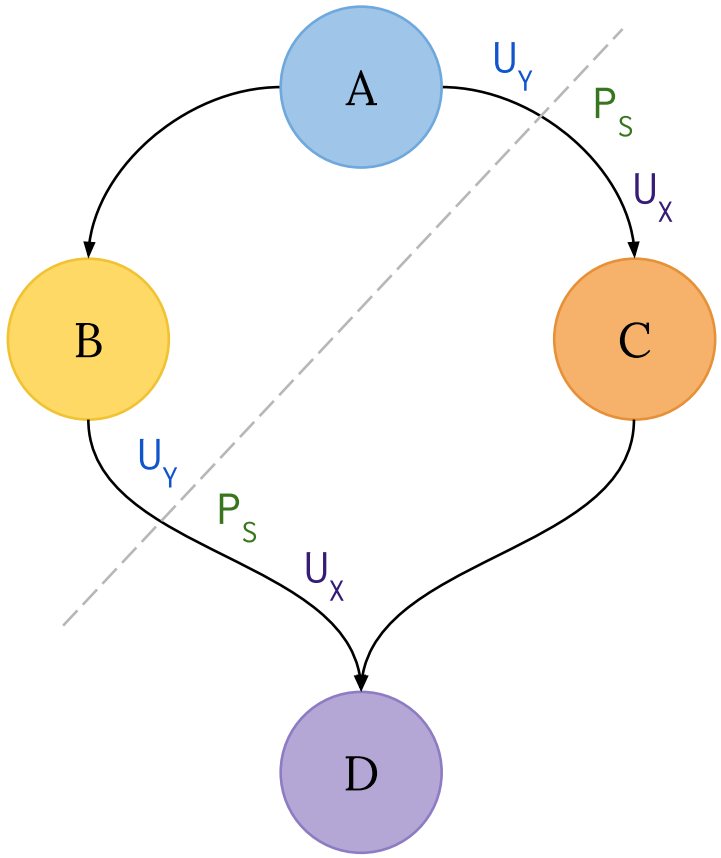

Figure 6.6: Updates propagate through the data-flow graph asynchronously. To correctly create a consistent snapshot, domain D should delay processing of U_Y until it has received the marker from both B and C.

Chapter 7

Evaluation

Neither persistent base nodes nor snapshotting lead to direct performance benefits for Soup. Instead, they both come with a wide range of features and improvements in other areas, further advancing Soup towards a production-ready storage system. Moving the base node state to durable storage lets Soup operate on datasets larger than its memory size, reduces its overall memory usage, and improves the system's overall recovery capabilities—a necessity if Soup is ever going to be usable for long running applications. Snapshotting takes this a step further, and removes the performance penalty Soup sees while its partial states are brought back to the state they were in prior to a fatal failure.

Regardless, Soup's performance when faced with a large amount of concurrent requests is still one of its main contributions. While a certain reduction in throughput and latency might be inevitable, it is crucial that this penalty remains as insignificant as possible. This chapter first investigates the effects `PersistentState` has on performance, followed by a look at the recovery benefits from both `PersistentState` and snapshotting.

7.1 Write-performance

Only a small subset of reads propagate all the way to the base nodes. Instead, they are served by partially materialized nodes further down the graph, avoiding the need for expensive computations on each read. This is not the case with writes. Every update that reaches Soup needs to be fully persisted to durable storage before a write acknowledgment can be sent. With persistent base nodes, that involves materializing the updates into `PersistentState`. With packets being processed synchronously at each domain, even the smallest increase in write latency at the base nodes could have disastrous effects for the overall write throughput of the system.

The `vote` benchmark described in section 4.3 is used to measure write-performance. While it is normally a mixed-load benchmark, where clients both write new votes and read the existing vote counts of articles, we will run `vote` with a pure write-load, removing reads altogether. Since we are measuring the impact of writing to durable storage, Soup will run without sharding, resulting in a single domain writing new votes to `PersistentState`. The database is prepopulated with 100K articles and the inserted votes are uniformly distributed across the existing articles.

Soup can be run both in a local and distributed fashion, and `vote` supports both. Writing to durable storage is a penalty fixed per machine however, and benchmarking its horizontal scalability makes little sense. Instead, we will use the *local* `vote` benchmark, where both the clients and the Soup workers run on the same machine.

The benchmark is run on the server described in section 4.1.1, with the write-ahead log written to one SSD and the database files to another.

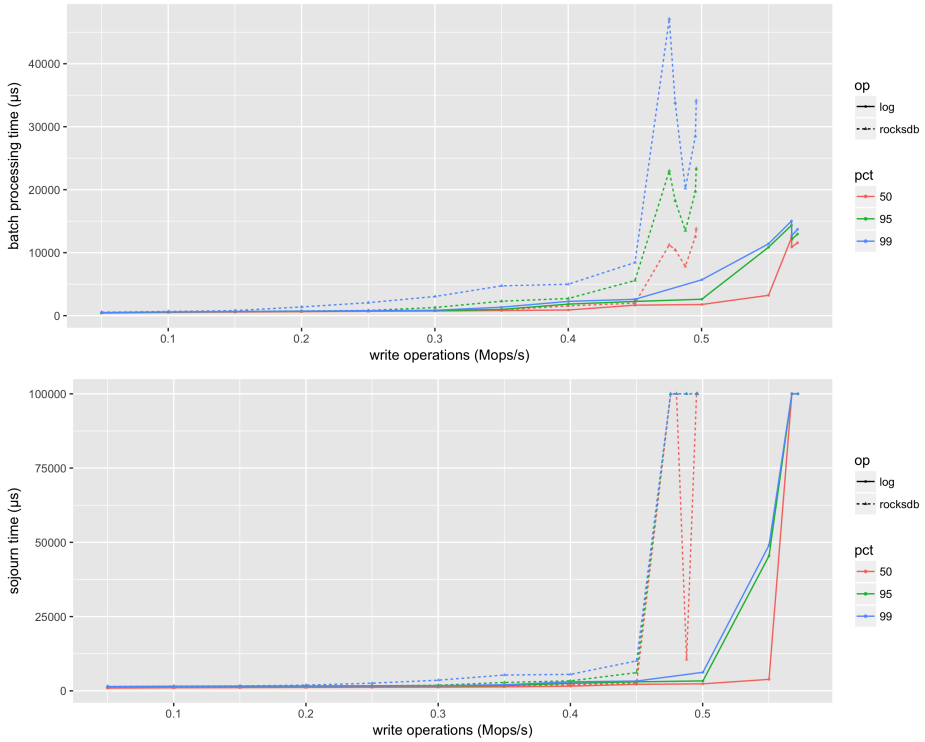

Figure 7.1: Write-only comparison of Soup's regular write-ahead log and RocksDB. The topmost figure shows the latency from when the request was initiated, while the bottommost includes the time from the request was generated by the open-loop benchmark (see section 4.3.1).

Materializing base node state to durable storage introduces a slight write-latency penalty under load. This eventually translates to about a 10% decrease in maximum write-throughput, compared to the naive Soup log. Even though the RocksDB write-ahead log is written to a different disk than its SS-tables, the amount of data that has to be written to the RocksDB WAL with `PersistentState` is still multiples more than with the Soup log. The former has to include entries for every index a base node maintains, while the latter only needs one entry for the update itself.

Additionally, writing to RocksDB's in-memory buffers is not a negligible cost, especially considering serialization. Whereas the Soup log only serializes updates once, `PersistentState` needs to do so, in-part, once for each index.

7.1.1 MemTable format

All writes to RocksDB are first placed in an in-memory buffer—a MemTable. RocksDB includes multiple MemTable implementations, and the advantages and disadvantages of the different data structures are described in section 2.3.9. To measure the impact of changing the MemTable implementation, we make use of the `vote` benchmark from the previous section, running on the same hardware.

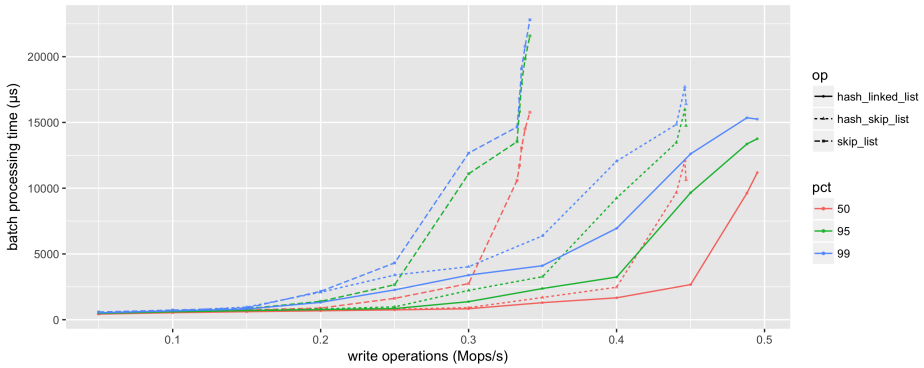

Figure 7.2: Write-latency measured at different throughput targets using the `vote` benchmark, while comparing separate RocksDB MemTable implementations.

Changing from the default skip list implementation results in a significant increase in write-throughput. In reality, this is a comparison of the $O(\log n)$ performance guarantees of a skip list and the closer to constant time guarantees of the hash-based implementations. RocksDB buckets keys based on the their prefix, where the two hash-based implementations use different data structures to organize each bucket. Using a skip list for the buckets result in $O(\log p)$ insertions, where p is the amount of keys within a single prefix. Linked list buckets improves this even further, with constant time insertions and lower memory overhead.

While the write performance of the link list hash table is better than the skip list one, its read performance is worse. The former requires a linear search to find values, while the latter can do the same in logarithmic time. Fortunately, RocksDB turns link list buckets into skip lists when they go beyond a certain amount of keys—by default 256. In the `vote` benchmark this will likely happen for the `Vote` base table, which will have a large index on `article_id`, but not for the `Article` table where each prefix maps to a single key, a unique article.

7.2 Read-performance

When measuring the overall read-performance of Soup as a system, a read-heavy `vote` benchmark is a good indicator. To analyze the impact of durable storage, we want to ensure that we are measuring the read performance of the base nodes, and not the partial nodes further down the graph. Instead, we will make use of the replay benchmark described in section 4.4, where each row is read at most once, resulting in a full replay from the base nodes.

The database is prepopulated with 10 million rows, after which a small random subset of the rows are read once. With `PersistentState`, Soup recovers existing data between each test run, after flushing the disk caches.

The benchmark is run on the same server as the write-throughput benchmark (see section 4.1.1).

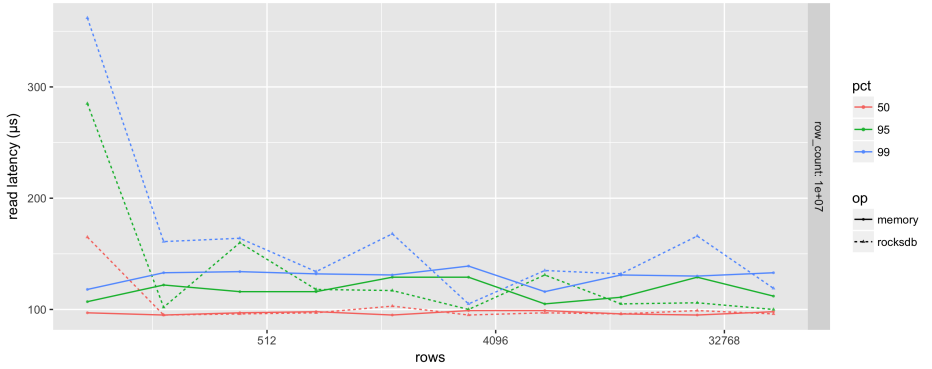

Figure 7.3: The replay performance of in-memory Soup compared to Soup with RocksDB.

Reading a small amount of rows result in less cache overlap when data is read from durable storage, which in turn leads to higher read latency. When the benchmark gravitates towards higher read counts, more data is bound to be served from the file system cache, reducing the overall latency of most reads.

7.2.1 SS-table format

RocksDB provides two separate SS-table implementations, `BlockBasedTable` and `PlainTable`, as described in section 2.3.10. The latter imposes limitations that the former does not have but promises lower read latency on fast storage mediums in return.

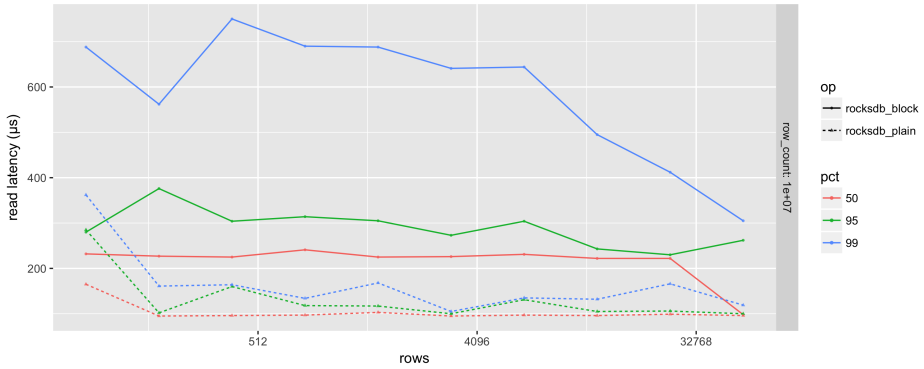

Figure 7.4: Soup replay performance comparison between `BlockBasedTable` and `PlainTable`.

Soup's persistence base tables rely heavily on prefix iteration—one of `PlainTable`'s design goals[1]. This results in significantly lower overall read latency compared to the traditional block-based format.

7.3 Mixed workload

The Lobsters benchmark described in section 4.2 is used to measure the performance of persistent bases in a real-world setting. The benchmark issues queries used in the real Lobsters web application, following a distribution modeled after the application's production traffic. Whereas the previous benchmarks measure queries per second, the Lobsters benchmark uses pages per second as its metric, where separate pages require different queries. A single page view can issue both read and write requests, through insertions, lookups, and updates. As Soup translates updates to removals, followed by insertions, the benchmark also measures deletion performance. The latency measured is the *sojourn* time of each request—the time from the request is generated until completion (see section 4.3.1).

The benchmark runs on two separate servers. SQL queries are issued by a workload generator and translated to native Soup requests using the MySQL shim layer (see section 2.1.6). These requests are handled by a separate server, running Soup. Before initiating the benchmark, the database is prepopulated with a similar amount of data to the real Lobsters application: 120,000 comments, 40,000 stories, and 9,200 users. Afterwards, the benchmark issues requests following a target throughput goal.

Note that unlike the write-throughput benchmark, Soup's write-ahead log is not used here. Instead, the benchmark compares purely in-memory—and not durable—Soup, to Soup with its base tables safely stored in durable RocksDB.

[1] `rocksdb.org/blog/2014/06/23/plaintable-a-new-file-format.html`

7.3.1 Computational overhead

The Lobsters benchmark is run on two separate hardware configurations. The first, server setup 2 (described in section 4.1.2), emulates durable storage using a RAM-disk[2]. This lets the benchmark focus on the computational overhead of storing Soup's base tables in RocksDB, while serving as a useful comparison to existing Soup benchmarks—which make use of the same server setup.

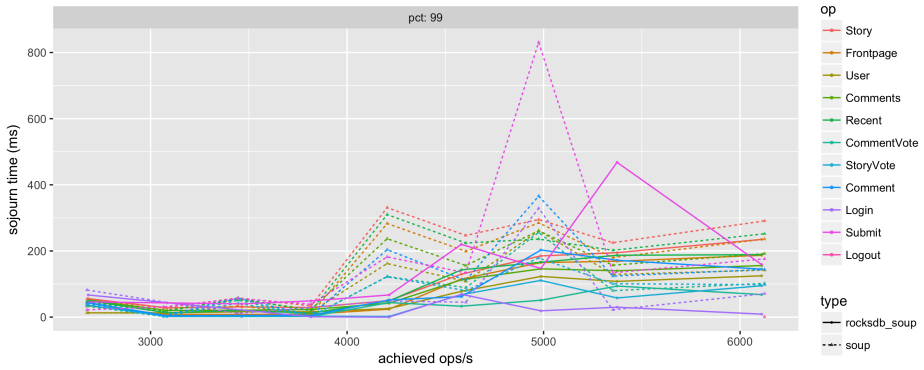

Figure 7.5: The 99th percentile sojourn latency of the Lobsters benchmark measured at increasing throughput. The `soup` target does not write any log files, while the `rocksdb_soup` target persists all updates to RocksDB.

Figure 7.5 shows that Soup with its base tables stored in RocksDB performs just as well as in-memory Soup. The seemingly random fluctuations in latency exist for both versions, and are not a result of whether the tables are stored in-memory or not.

7.3.2 I/O overhead

To measure the overhead of having to persist updates to a durable write-ahead log, the same benchmark is run on an Amazon EC2 server with an NVMe SSD-drive[3] (server setup 2 in section 4.1.2). Similar to the write-throughput benchmark in section 7.1, the write-ahead log and RocksDB database files are written to separate disks.

[2] https://manpages.debian.org/stretch/initscripts/tmpfs.5.en.html
[3] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ssd-instance-store.html

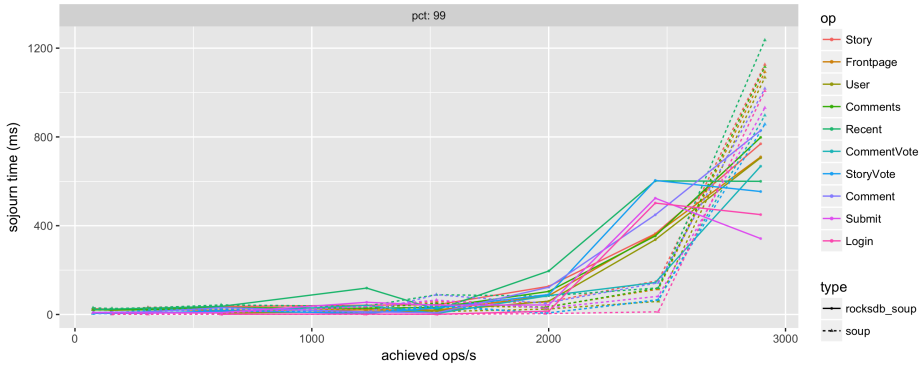

Figure 7.6: 99th percentile sojourn latency of the Lobsters benchmark measured at increasing throughput. The `rocksdb_soup` target writes to a durable NVMe SSD.

The observed latency is slightly higher when Soup's base tables are stored on durable storage, but the difference is far from significant. The server used here is on the other hand far slower than the one in the previous section, resulting in overall lower throughput targets. While durable and non-durable Soup are quite equal here as well, the difference would without doubt be more drastic on a server with slower storage media.

7.4 Recovery

The recovery benchmark introduced in section 4.5 helps us compare the recovery impact imposed by different durability strategies. For every data point, the database is populated with 10K articles and a varying amount of votes divided evenly between the articles. After population, Soup is restarted, while the time it takes to recover is measured. The state is considered recovered when the total sum of votes returned from reading all articles equal the actual amount of votes in the system—signified as *total* in figure 7.7.

Unlike the other durability methods, recovering with durable base nodes does not affect the partial nodes further down the graph—they remain empty until future read operations trigger ancestor queries to the base nodes. Snapshotting, on the other hand, brings all materialized nodes in the graph back to the state they were in prior to crashing. This is the case for log-recovery as well, as updates from the WAL propagate through the entire graph when recovering. To highlight this divide, the time it takes to read a single key after recovering from a durable base node application is measured as well, denoted as *initial* in figure 7.7.

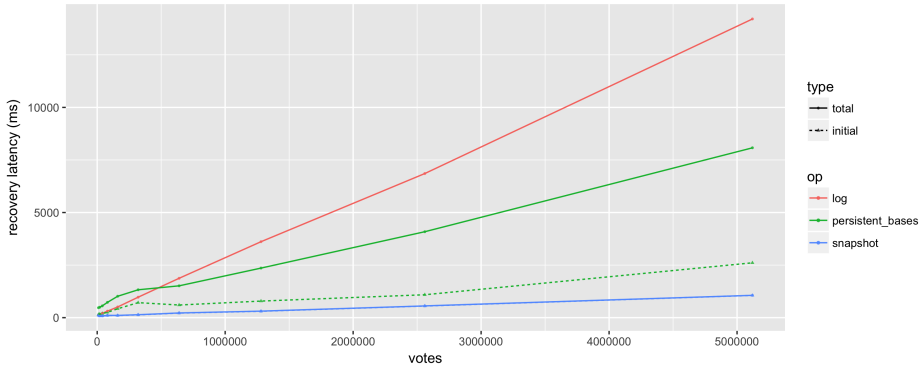

Figure 7.7: The recovery benchmark measures the time it takes to recover after a failure. The *initial* metric highlights the latency of reading a single key, while *total* requires all reads to return the same result as prior to crashing.

The results are pretty much as expected. Snapshotting returns all the materialized nodes in the graph to their correct state, avoiding the need to replay any state after recovering. The time it takes to recover still increases after a while, with more data to read from disk. Log-based recovery needs to go through *all* updates since the beginning of time before it is considered ready, resulting in poor performance. With durable base nodes, each read requires a full replay from the bases—a significant latency penalty when the row count goes up.

With `PersistentState`, the base nodes do not have to process any updates at all when recovering. Restoring `PersistentState` to the correct state is left to RocksDB, which puts a cap on recovery time by ensuring that its write-ahead logs never grow beyond a given size. Recovering the actual database files, the SS-tables, is "free"—no data needs to be read into memory.

7.4.1 Snapshot compression

Compressing snapshots introduce a trade-off between computation and storage overhead. Depending on the underlying hardware, the penalty induced from having to compress and decompress snapshots might be amortized by the reduced I/O usage. To measure the impact on recovery performance, the benchmark used in the previous section is run on the same hardware—with and without `zlib` compression.

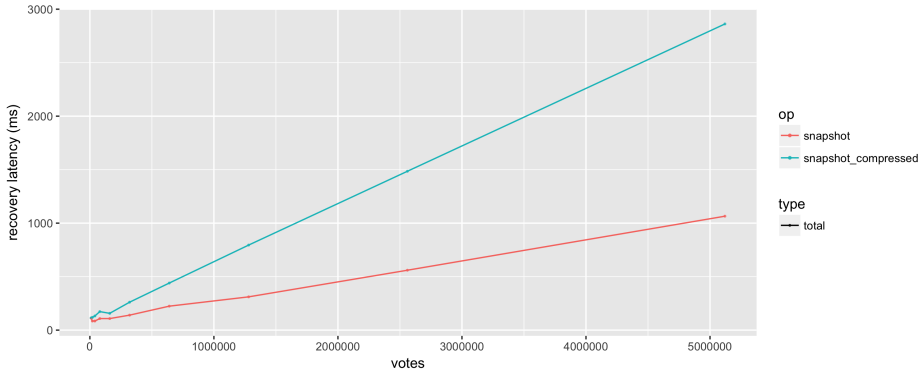

Figure 7.8: The total snapshot recovery time, with and without compression.

Recovering from compressed snapshots takes on average three times as long as recovering from regular snapshots. This is measured on a server with a SSD-drive, and the gap would without doubt be less significant on slower storage media. Regardless, with a focus on reducing recovery time, this is not a trade-off we are willing to make.

7.4.2 Write-performance with snapshotting

Snapshotting is a significant improvement to Soup's recovery situation and a step in the right direction for Soup as a production-ready system. Regardless, it is only useful if Soup manages to maintain much of the same write-throughput while performing regular snapshots.

To measure the performance penalty of snapshotting, we make use of the `vote` benchmark described in section 4.3 and earlier in this chapter. The benchmark compares the batch write latency at increasing throughput targets, both with and without snapshotting. The benchmark runs for 60 seconds, performing a snapshot every 10 seconds.

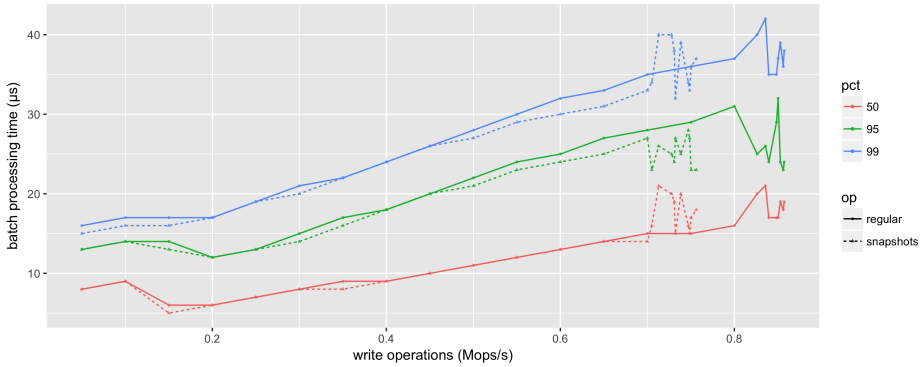

Figure 7.9: Write-latency comparison with and without snapshotting (with a snapshot timeout of 10 seconds). Both use Soup's regular write-ahead log.

Most of the snapshotting work is performed in standalone snapshot workers, running in threads separate from Soup's packet processing. Without this, the throughput penalty would without doubt be much more significant than the 10% observed in figure 7.9. The penalty is a result of the full state clone incurred at each materialized node during a snapshot.

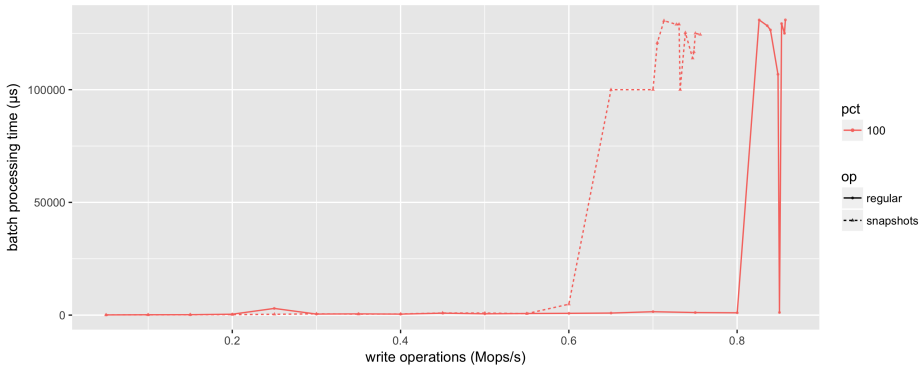

Figure 7.10: 100th percentile comparison, with and without snapshotting.

The first snapshot graph shows no increase in latency. With a snapshot timeout of 10 seconds and a benchmark runtime of 60 seconds, snapshot occurrences are probably too rare for it to show up in the 95th percentile. Looking at the 100th percentile on the other hand, we can see the latency spiking at a lower throughput than normal. At this point the state size is reasonably large, resulting in non-trivial clone operations.

Chapter 8

Conclusion

This chapter looks back at the results presented earlier in this thesis, drawing conclusions from both the positive and the negative side-effects of the contributed implementations. Afterwards, it takes a look at possible durability directions Soup might take in the future, towards the eventual goal of becoming a production-ready system.

8.1 Persistent base tables

Storing base tables on durable storage, instead of in volatile memory, lets Soup handle continuously increasing quantities of data without having its memory usage grow without bounds. With base tables stored safely on durable storage, Soup no longer needs to replay all previous updates to recover after a failure. Read requests instead replay data from the base tables when needed, eventually bringing Soup's partially materialized views back to a similar state they were in prior to the failure.

The persistent base table implementation contributed throughout this thesis replaces Soup's in-memory index structure with a durable index structure built on top of RocksDB—a battle-tested key-value storage layer developed at Facebook. Compared to Soup's previous ever-growing write-ahead log, the RocksDB-based implementation decreases the raw write throughput with about 10%, while moving Soup from a pure main-memory database to a storage layer capable of handling data larger than its resident memory size, with significant benefits for both recovery and Soup's overall memory usage. The implementation is currently used by the latest Soup prototype.

8.2 Snapshotting

With base tables persisted to durable storage, Soup can begin serving requests shortly after recovering from a failure. Unfortunately, these requests will need to go through the entire data-flow graph before returning, with Soup's partially materialized views starting out empty after recovering. Akin to the performance penalty induced by a cold cache, Soup performs read requests with higher latency until the application's working set again is present in the partially materialized nodes throughout the graph. Snapshotting avoids this by restoring all materialized state, after which Soup can continue serving read operations as if nothing happened.

Compared to log-based recovery, the snapshotting implementation described throughout this thesis recovers in less than a tenth of the time, at the cost of a 10% overall penalty to Soup's write throughput. While rapid recovery is an important property for a database system, the performance penalty induced here comes with considerably less benefits than moving Soup's base tables to durable storage, leaving the question of whether it is a reasonable compromise open. At the same time, the effects of snapshotting over a longer period of time have yet to be investigated and should be considered—especially in regards to the proposed improvements to the current snapshotting method. All in all, snapshotting is a promising concept, with the possibility of being an important part of a production-ready Soup system in the future.

8.3 Conclusion

This thesis presents the internals of the Soup structured storage system and the challenges faced by its current solution for maintaining write durability. Prior to the contributions described throughout this thesis, Soup was a pure main-memory database, incapable of handling datasets beyond the size of its available memory. Now, Soup stores the majority of its data in durable index structures, while maintaining much of the same performance guarantees as before. Finally, Soup recovers considerably faster after failures, as it no longer has to go through and re-apply all updates from its previously ever-growing write-ahead log.

8.4 Future work

This section presents possibly paths of improvement for the two main contributions presented throughout this thesis.

8.4.1 Snapshotting and persistent bases

The `PersistentState` implementation described in section 5 removes the regular Soup write-ahead log in favor of relying on RocksDB for durability. RocksDB maintains its own WAL, which, unlike the Soup WAL, is discarded when its updates are safely flushed to durable storage. This is far better for recovery purposes, as it avoids the need to go through a seemingly endless stream of updates to restore Soup back to a pre-failure state. It does, on the other hand, complicate matters for snapshotting.

Snapshotting relies on Soup's write-ahead log to recover updates that occur after a snapshot is taken, prior to a failure. During recovery, the latest snapshot is first restored, followed by log-based recovery for any remaining log entries. Together they make sure that Soup recovers quickly, without degrading its durability guarantees. With persistent bases the write-ahead log is maintained internally by RocksDB, together with the decision of when to eventually discard prior log files. Without the ability to replay log entries, recovering using snapshotting would leave all other nodes than the base nodes in an older state than before the crash.

Recovery using persistent bases leaves the partial nodes further down the graph empty. This works fine because of Soup's replay system: any missing reads will propagate all the way to the base nodes, resulting in the partial nodes eventually reaching a similar state to the one they were in prior to crashing. With snapshotting, the partial nodes would end up in an *old* state, instead of empty. This would prevent Soup from issuing base node replays, effectively discarding the updates that happened after the last snapshot was taken.

That leaves the question of how to replay updates that happened after the last snapshot was taken, while still relying on RocksDB's write-ahead log for persistence. The first

step would be to ensure that RocksDB never discards WAL files until all its updates are included in a snapshot. Secondly, Soup's recovery procedure would need to retrieve updates that happened after the last snapshot was taken, directly from the RocksDB write-ahead logs. By including the current snapshot identifier in all persisted updates, the recovery process would be able to discern between updates that happened before and after the last persisted snapshot.

8.4.2 `PersistentState` serialization

Both the keys and values persisted to RocksDB are serialized using bincode (see section 2.5). While bincode performs well compared to other serialization libraries, implementing encoding techniques specifically for Soup's use case would come with other potential benefits. One example is specific sorting orders in `PersistentState`. With the current implementation, keys would have to be deserialized before they could be compared to each other, resulting in unnecessary allocations. These allocations would be avoided by using an encoding scheme where keys could be compared without deserialization, such as *e.g.*, MyRock's `memcomparable` format [32].

8.4.3 Uncoordinated snapshots

Coordinating a global snapshot across the entire data-flow graph requires unnecessary communication between the workers and the controller. Instead, the question of finding the last valid snapshot could be left to the recovery process, *e.g.*, by finding $Min(epoch)$ across the nodes, or by following schemes such as [43].

8.4.4 Incremental snapshots

The write-performance benchmark in section 7.4.2 showed a 10% decrease in overall write throughput after introducing snapshotting. The majority of the work is performed in separate snapshotting threads, leaving the state clone operation as the culprit. To avoid cloning altogether, snapshots would need to be maintained gradually, which could be achieved by maintaining a buffer of changes between snapshots, which could then be forwarded to the snapshotting worker and applied there. While this avoids the need to clone the entire state, snapshot workers would now need to keep an entirely duplicate clone of the snapshot state in memory, effectively doubling Soup's memory usage.

Instead, snapshots could be maintained incrementally directly on durable storage. This could make use of the same `PersistentState` implementation used by persistent base nodes, either by having the snapshotting workers apply received updates to RocksDB, or by doing so directly from each domain. This would significantly reduce the write-amplification required to persist a snapshot, by avoiding the need to write duplicate data to disk again and again. Incremental snapshotting would also minimize the risk of filling up the snapshot workers' queues, which could now happen if the time it takes

to serialize and persist a single, possibly large, snapshot grows beyond the predefined snapshot interval.

Appendix A

Contributions

The implementations described throughout this thesis have resulted in a series of contributions to various open-source projects. This appendix lists a selected subset of the *pull requests* contributed to these projects, where a pull request is simply a proposed unit of changes.

A.1 distributary

distributary is the prototype implementation of Soup, written in Rust.

Title	Pull request
RocksDB Persistence	https://git.io/vhf7w (#72)
SQLite Base Node Indices	https://git.io/vhf7B (#58)
Initial Snapshotting Implementation	https://git.io/vhf79 (#54)
Recovery	https://git.io/vhfbI (#37)
Arithmetic Expressions in Projections	https://git.io/vhf7F (#35)
AUTO_INCREMENT support	https://git.io/vhf7b (#66)
Refactor LookupResult	https://git.io/vhf5f (#76)
Remove generics from State	https://git.io/vhf5k (#64)
Materialization status in graphviz	https://git.io/vhf5L (#51)
Use appropriate names for literals and arithmetic expressions	https://git.io/vhf5m (#45)
Add tests for Extremum::MIN	https://git.io/vhf5G (#33)

A.2 distributary-mysql

distributary-mysql is the MySQL protocol translation layer described in section 2.1.6.

Title	Pull request
UPDATE and DELETE support	https://git.io/vhfdi (#12)

A.3 nom-sql

`nom-sql` is the SQL parser used by `distributary` and `distributary-mysql`, written in Rust.

Title	Pull request
Add aliases to arithmetic expression	https://git.io/vhf5D (#8)
Upgrade `nom`	https://git.io/vhfdv (#13)
Move alias up to `FieldExpression`	https://git.io/vhf5h (#15)
Attach the table name to keys and columns in `CreateTableStatement`	https://git.io/vhfdZ (#16)
Implement `fmt::Display` for `ArithmeticExpression`	https://git.io/vhf5F (#12)

A.4 RocksDB

`RocksDB` is the key-value store described in section 2.3, which the durability layer in Soup is implemented on top of.

Title	Pull request
Add manual WAL flushing to the C API	https://git.io/vhfb8 (#3792)

A.5 rust-rocksdb

rust-rocksdb is a Rust wrapper library for RocksDB.

Title	Pull request
Add support for customizing the memtable factory	https://git.io/vhfAV (#180)
Support linking to other compression libraries	https://git.io/vhfAg (#185)
Add set_memtable_prefix_ratio	https://git.io/vhfAE (#181)
Make sure DB is dropped after all tests	https://git.io/vhfAB (#183)
Add index_type customization to BlockBasedOptions	https://git.io/vhfAl (#184)
Add DBOptions.set_wal_dir	https://git.io/vhfN8 (#186)
Add disable_cache method to BlockBasedOptions	https://git.io/vhfNC (#188)

Bibliography

[1] A. Acharya and B. R. Badrinath. “Recording Distributed Snapshots Based on Causal Order of Message Delivery”. In: *Inf. Process. Lett.* 44.6 (Dec. 1992), pages 317–321.

[2] S. Alagar and S. Venkatesan. “An Optimal Algorithm for Distributed Snapshots with Causal Message Ordering”. In: *Inf. Process. Lett.* 50.6 (June 1994), pages 311–316.

[3] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. “Storage Management in AsterixDB”. In: *Proc. VLDB Endow.* 7.10 (June 2014), pages 841–852.

[4] Apple. *FoundationDB - the open source, distributed, transactional key-value store.* Apr. 2018. URL: https://github.com/apple/foundationdb (visited on 03/17/2018).

[5] J. Arulraj, M. Perron, and A. Pavlo. “Write-behind Logging”. In: *Proc. VLDB Endow.* 10.4 (Nov. 2016), pages 337–348.

[6] D. F. Bacon et al. “Spanner: Becoming a SQL System”. In: *Proceedings of the 2017 ACM International Conference on Management of Data.* SIGMOD ’17. Chicago, Illinois, USA: ACM, 2017, pages 331–343.

[7] R. Bayer and E. McCreight. “Organization and Maintenance of Large Ordered Indices”. In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control.* SIGFIDET ’70. Houston, Texas: ACM, 1970, pages 107–141.

[8] P. A. Bernstein and N. Goodman. “Concurrency Control in Distributed Database Systems”. In: *ACM Comput. Surv.* 13.2 (June 1981), pages 185–221.

[9] B. H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Commun. ACM* 13.7 (July 1970), pages 422–426.

[10] M. Callaghan. *Read, write & space amplification — B-Tree vs LSM*. 2015. URL: http://smalldatum.blogspot.com/2015/11/read-write-space-amplification-b-tree.html (visited on 04/05/2018).

[11] K. M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Trans. Comput. Syst.* 3.1 (Feb. 1985), pages 63–75.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pages 205–218.

[13] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. "From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pages 197–212.

[14] CockroachDB. *CockroachDB*. Jan. 2018. URL: https://www.cockroachlabs.com/ (visited on 04/20/2018).

[15] CockroachDB. *CockroachDB Design*. Mar. 2018. URL: https://github.com/cockroachdb/cockroach/blob/master/docs/design.md (visited on 04/10/2018).

[16] CockroachDB. *Structured data encoding in CockroachDB SQL*. Jan. 2018. URL: https://github.com/cockroachdb/cockroach/blob/master/docs/tech-notes/encoding.md (visited on 04/20/2018).

[17] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (June 1970), pages 377–387.

[18] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pages 1277–1288.

[19] J. C. Corbett et al. "Spanner: Google's Globally Distributed Database". In: *ACM Trans. Comput. Syst.* 31.3 (Aug. 2013), 8:1–8:22.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pages 205–220.

[21] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. *Implementation techniques for main memory database systems*. Volume 14. 2. ACM, 1984.

[22] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. "Hekaton: SQL Server's Memory-optimized OLTP Engine". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Man-*

agement of Data. SIGMOD '13. New York, New York, USA: ACM, 2013, pages 1243–1254.

[23] J. V. D'silva, R. Ruiz-Carrillo, C. Yu, M. Y. Ahmad, and B. Kemme. "Secondary Indexing Techniques for Key-Value Stores: Two Rings To Rule Them All". In: *Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, March 21-24, 2017*. 2017.

[24] M. H. Eich. "Parallel Architectures for Database Systems". In: edited by A. R. Hurson, L. L. Miller, and S. H. Pakzad. Piscataway, NJ, USA: IEEE Press, 1989. Chapter A Classification and Comparison of Main Memory Database Recovery Techniques, pages 417–424.

[25] L. M. B. Ek. "Efficient recovery in a data-flow based storage system". TDT4501 Computer Science, Specialization Project. Dec. 2017.

[26] R. Escriva, B. Wong, and E. G. Sirer. "HyperDex: A Distributed, Searchable Key-value Store". In: *SIGCOMM Comput. Commun. Rev.* 42.4 (Aug. 2012), pages 25–36.

[27] Facebook. *RocksDB Iterator*. Mar. 2017. URL: https://github.com/facebook/rocksdb/wiki/Iterator (visited on 02/23/2018).

[28] Facebook Open Source. *Leveled Compaction*. Aug. 2017. URL: https://github.com/facebook/rocksdb/wiki/Merge-Operator (visited on 03/20/2018).

[29] Facebook Open Source. *A persistent key-value store for fast storage environments*. Apr. 2018. URL: http://rocksdb.org/ (visited on 04/20/2018).

[30] Facebook Open Source. *Leveled Compaction*. Apr. 2018. URL: https://github.com/facebook/rocksdb/wiki/Leveled-Compaction (visited on 04/20/2018).

[31] Facebook Open Source. *MyRocks*. Apr. 2018. URL: http://myrocks.io/ (visited on 04/20/2018).

[32] Facebook Open Source. *MyRocks record format*. Apr. 2018. URL: https://github.com/facebook/mysql-5.6/wiki/MyRocks-record-format (visited on 04/20/2018).

[33] Facebook Open Source. *Write Ahead Log Format*. Apr. 2018. URL: https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log-File-Format (visited on 04/20/2018).

[34] S. Finkelstein. "Common Expression Analysis in Database Applications". In: *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*. SIGMOD '82. Orlando, Florida: ACM, 1982, pages 235–245.

[35] J. Gallagher. *Rusqlite*. [Online; accessed 3-March-2018]. May 2018. URL: https://github.com/jgallagher/rusqlite.

[36] GitHub. *GitHub's online schema migration for MySQL*. May 2018. URL: https://github.com/github/gh-ost (visited on 02/02/2018).

[37] J. Gjengset. *evmap*. URL: https://github.com/jonhoo/rust-evmap (visited on 02/14/2018).

[38] J. Gjengset. "Xylem: flexible and high-performance structured storage via dynamic data-flow". In: SOSP Student Research Competition. 2017.

[39] Google. *LevelDB Iteration*. Mar. 2017. URL: https://github.com/google/leveldb/blob/master/doc/index.md#iteration (visited on 02/23/2018).

[40] T. Haerder and A. Reuter. "Principles of Transaction-oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pages 287–317.

[41] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. "OLTP Through the Looking Glass, and What We Found There". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pages 981–992.

[42] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, pages 11–11.

[43] M. Isard and M. Abadi. "Falkirk Wheel: Rollback Recovery for Dataflow Systems". In: *CoRR* abs/1503.08877 (2015). arXiv: 1503.08877.

[44] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. "DalÍ: A High Performance Main Memory Storage Manager". In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pages 48–59.

[45] R. Kallman et al. "H-store: A High-performance, Distributed Main Memory Transaction Processing System". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pages 1496–1499.

[46] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. "SLIK: Scalable Low-Latency Indexes for a Key-Value Store". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pages 57–70.

[47] A. Kemper and T. Neumann. "HyPer: A hybrid OLTP amp;OLAP main memory database system based on virtual memory snapshots". In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pages 195–206.

[48] E. Kohler. *HotCRP Conference Review Software*. May 2018. URL: https://github.com/kohler/hotcrp (visited on 05/20/2018).

[49] A. D. Kshemkalyani, M Raynal, and M Singhal. "An introduction to snapshot algorithms in distributed computing". In: *Distributed Systems Engineering* 2.4 (1995), page 224.

[50] T. H. Lai and T. H. Yang. "On Distributed Snapshots". In: *Inf. Process. Lett.* 25.3 (May 1987), pages 153–158.

[51] L. Lamport. “The Part-time Parliament”. In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pages 133–169.

[52] A. P. Liedes and A. Wolski. “SIREN: A Memory-Conserving, Snapshot-Consistent Checkpoint Algorithm for in-Memory Databases”. In: *22nd International Conference on Data Engineering (ICDE'06)*. 2006, pages 99–99.

[53] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. “Existential Consistency: Measuring and Understanding Consistency at Facebook”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pages 295–310.

[54] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. “Rethinking main memory OLTP recovery”. In: *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE. 2014, pages 604–615.

[55] F. Mattern. “Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation”. In: *J. Parallel Distrib. Comput.* 18.4 (Aug. 1993), pages 423–434.

[56] J. Mayo and P. Kearns. “Efficient Distributed Termination Detection with Roughly Synchronized Clocks”. In: *Parallel and Distributed Computing and Systems*. IASTED/ACTA Press, 1995, pages 305–307.

[57] C. G. Mike Ray, G. Milener, S. Stein, and B. Kess. *Heaps (Tables without Clustered Indexes)*. 2016. URL: https://docs.microsoft.com/en-us/sql/relational-databases/indexes/heaps-tables-without-clustered-indexes?view=sql-server-2017 (visited on 03/05/2018).

[58] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. “ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging”. In: *ACM Trans. Database Syst.* 17.1 (Mar. 1992), pages 94–162.

[59] MySQL. *Appendix B InnoDB Source Code Distribution*. 2018. URL: https://dev.mysql.com/doc/internals/en/files-in-innodb-sources.html (visited on 03/05/2018).

[60] R. Nishtala et al. “Scaling Memcache at Facebook”. In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pages 385–398.

[61] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. “The Log-structured Merge-tree (LSM-tree)”. In: *Acta Inf.* 33.4 (June 1996), pages 351–385.

[62] D. Ongaro and J. Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pages 305–320.

[63] T. Overby. *bincode*. [Online; accessed 6-February-2018]. May 2018. URL: https://github.com/TyOverby/bincode.

[64] M. PDOS. *MySQL/MariaDB protocol shim for Soup*. Feb. 2018. URL: https://github.com/mit-pdos/distributary-mysql (visited on 02/04/2018).

[65] PingCAP. *TiDB*. Apr. 2018. URL: https://github.com/pingcap/tidb (visited on 04/20/2018).

[66] M. A. Qader, S. Cheng, and V. Hristidis. "A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pages 551–566.

[67] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson. "Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pages 1539–1551.

[68] G. L. Sanders and S. Shin. "Denormalization effects on performance of RDBMS". In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. 2001, 9 pp.–.

[69] B. Schroeder, A. Wierman, and M. Harchol-Balter. "Open Versus Closed: A Cautionary Tale". In: *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*. NSDI'06. San Jose, CA: USENIX Association, 2006, pages 18–18.

[70] A. Scotti et al. "Comdb2 Bloomberg's Highly Available Relational Database System". In: *Proc. VLDB Endow.* 9.13 (Sept. 2016), pages 1377–1388.

[71] L. Shen. *TiDB Internal (II) - Computing*. 2017. URL: https://pingcap.com/blog/2017-07-11-tidbinternal2/#sqlonkv (visited on 03/03/2018).

[72] SQLite. *SQLite*. Apr. 2018. URL: https://www.sqlite.org (visited on 03/04/2018).

[73] SQLite. *Write-Ahead Logging*. Apr. 2018. URL: https://www.sqlite.org/wal.html (visited on 03/04/2018).

[74] M. Stonebraker et al. "C-store: A Column-oriented DBMS". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pages 553–564.

[75] H. Suzuki. *The Internals of PostgreSQL*. 2018. URL: http://www.interdb.jp/pg/pgsql01.html (visited on 03/05/2018).

[76] A. Tai, M. Wei, M. J. Freedman, I. Abraham, and D. Malkhi. "Replex: AScalable, Highly Available Multi-Index Data Store". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pages 337–350.

[77] Y. Tang, A. Iyengar, W. Tan, L. Fong, L. Liu, and B. Palanisamy. "Deferred Lightweight Indexing for Log-Structured Key-Value Stores". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2015, pages 11–20.

[78] The Apache Software Foundation. *Apache HBase*. 2018. URL: https://hbase.apache.org/ (visited on 06/01/2018).

[79] W. Vogels. *Eventually Consistent, Revisited*. 2008. URL: https://www.allthingsdistributed.com/2008/12/eventually_consistent.html (visited on 03/10/2018).

[80] J. Xu. *Online migrations at scale*. Feb. 2017. URL: https://stripe.com/blog/online-migrations (visited on 02/02/2018).

[81] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. "Performance Analysis of NVMe SSDs and Their Implication on Real World Databases". In: *Proceedings of the 8th ACM International Systems and Storage Conference*. SYSTOR '15. Haifa, Israel: ACM, 2015, 6:1–6:11.