ABSTRACT

Title of Dissertation: The Evolution of Cloud Data Architectures: Storage, Compute, and Migration

Gang Liao
Doctor of Philosophy, 2022

Dissertation Directed by: Professor Daniel J. Abadi
Department of Computer Science

Recent advances in data architectures have shifted from on-premises to the cloud. However, new challenges emerge as data explosion continues to expand at an exponential rate. As a result, my Ph.D. research focuses on addressing the following challenges.

First, cloud data warehouses such as Snowflake [1, 2], BigQuery [3], Redshift [4] often rely on storage systems such as distributed file systems or object stores to store massive amounts of data. The growth of data volumes is accompanied by an increase in the number of objects stored and the amount of metadata such systems must manage. By treating metadata management similar to data management, we built FileScale, an HDFS-based file system that replaces metadata management in HDFS with a three-tiered distributed architecture that incorporates a high throughput, distributed main-memory database system at the lowest layer, along with distributed caching and routing functionality above it. FileScale performs comparably to the single-machine architecture at a small scale,

while enabling linear scalability as the file system metadata increases.

Second, Function as a Service, or FaaS, is a serverless model that allows users decompose their applications into short-lived cloud functions. FaaS provides more fine-grained elasticity with sub-second start-up times that can dynamically match the per-query basis with continuous scaling. Customers are only charged for the execution time they consume, often at a granularity of one millisecond. To explore the promise of function services for stream processing, we built Flock, a cloud-native streaming query engine that runs on FaaS platforms. Flock is low cost to operate under low demand and can scale automatically to a high load at a proportional cost.

Third, Software as a Service, or SaaS, is a method of software product delivery to end-users over the internet and via pay-as-you-go pricing in which the software is centrally hosted and managed by the cloud service provider. Continuous Deployment (CD) in SaaS, an aspect of DevOps, is the increasingly popular practice of frequent, automated deployment of software changes. To realize the benefits of CD, it must be straightforward to deploy updates to both front-end code and the database, even when the database's schema has changed. Unfortunately, this is where current practices run into difficulty. So we built BullFrog, a PostgreSQL extension that is the first system to use lazy schema migration to support single-step, online schema evolution without downtime, which achieves efficient, exactly-once physical migration of data under contention.

The Evolution of Cloud Data Architectures: Storage, Compute, and Migration

by

Gang Liao

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:
Professor Daniel Abadi, Chair/Advisor
Professor Amol Deshpande
Professor Michael Hicks
Professor Louiqa Raschid
Associate Professor Pete Keleher

Acknowledgments

My Ph.D. journey leaves me with wonderful great memories and experiences on research and life. Looking back to the past four years, I want to thank many people for their support and encouragement.

First and foremost, I would like to express my gratitude to my advisor, Daniel Abadi, an excellent research mentor and a remarkable person who has made my graduate study a precious and enjoyable experience. Dan allows me to work on a wide range of research topics, broadening my horizons and deepening my understanding of the data system field. He listens patiently to my immature ideas and continually encourages me to tackle open problems involving technical innovation. His sincerity to the research work impacts me to conduct solid research, which helps me construct my research style and shapes my personality into the way I wish I could be. I feel lucky to have had Dan accompanying me walking through this Ph.D. journey.

Thank you, Amol Deshpande, Michael Hicks, Pete Keleher, and Louiqa Raschid, for serving as my committee members. I appreciate their time for reading my dissertation and providing insightful feedback for helping me improve the quality of my dissertation. My Ph.D. work was done in collaboration with Amol and Michael. Without their consistent help in formulating ideas and writing papers,

the thesis would be far from being completed.

I am fortunate to work with many fantastic mentors and peers who contributed their time and insights. Thank you, Souvik Bhattacherjee, for encouraging me to take over the research project despite my little relevant experience. We worked together even when I spent nearly a year in China struggling with a travel ban during the pandemic. I am also fortunate to have had multiple brilliant research mentors from internships. Thank you, Yinan Li, Badrish Chandramouli, and Donald Kossmann from Microsoft Research, for their valuable comments and discussions, which helped frame the way I conduct research. Their serious attitude to the research, the habit of seeking the root cause, and the system way for results analysis impact my research style. Thank you, Chang Lan and Chuanxiong Guo, for my first Ph.D. internship at ByteDance AI Lab. Thank you, Zejun Liu, Dennis Li, Zhichao Liu, and Zhenghang Hu, for participating and contributing to my Ph.D. research projects. I especially thank Yanfang Le and Rui Guo for sharing their Ph.D. experience and research ideas.

Thank you, Bingfa Li, who introduced me to the systems research field, serving as my undergraduate research advisor. Thanks to his mentorship, I started along the path to where I am today. Thank you, Markus Hadwiger, David Keyes, and Yi Wang, for understanding and supporting my return from industry to academia for my Ph.D. study.

Finally, I am incredibly grateful to my parents and sisters. They provide me with their support in all possible ways. My heartfelt thanks to my wife, Jane, who gives me her unlimited support and encouragement, making my everyday life

much more colorful and joyful, and to my three-year-old son, Jimmy, who brings me tons of memorable moments. This dissertation would not be possible without their constant love, support, and encouragement.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CD	Continuous Deployment
TPC-C	Transaction Processing Performance Council Benchmark C
DDL	Data Definition Language
UDF	User-defined Function
SQL	Structured Query Language
OLTP	Online Transaction Processing
OLAP	Online Analytical Processing
TPS	Transactions per Second
ACID	Atomicity, Consistency, Isolation, Durability
TTL	Time to Live
HA	High Availability
SLO	Service-level Objective
SLA	Service-level Agreement
YSB	Yahoo Streaming Benchmarks
SIMD	Single Instruction/Multiple Data
DAG	Directionally-acyclic Graph
UUID	A Universally Unique Identifier
DLQ	Dead-letter Queue
FaaS	Function as a Service
SaaS	Software as a service

Chapter 1: Introduction

1.1 Overview

The cloud benefits have driven many recent efforts to port data analytics from on-premises to cloud. The migration of workloads has led in a fast expansion of cloud services, resulting in significant innovation and new research problems. In this dissertation, we aim to address the following question: How can cloud infrastructure be used to rethink the next-generation cloud data systems to tackle **large-scale storage**, **computation**, and **data migration** concerns as the data explosion proceeds at an exponential rate? We built three systems: FileScale, Flock, and Bullfrog to study and tackle the difficulties of the cloud data architecture, including scalability of storage, fine-grained elasticity of computing, and exactly-once online schema evolution. We now describe each of these works in more detail in the following sections.

1.2 FileScale

Cloud data warehouses such as Snowflake [1, 2], BigQuery [3], Redshift [4] often rely on storage systems such as distributed file systems or object stores to

store massive amounts of data. The growth of data volumes is accompanied by an increase in the number of unstructured byte contents of the files stored and the amount of the structured metadata such systems must manage. In general, it is easier to scale the unstructured data than the structured data, since there is no requirement to perform atomic transactions that update the unstructured bits across multiple files.

However, scaling the structured data is more challenging for several reasons: First, there is a requirement for ACID transactions that may access data in multiple partitions. For example, recursively deleting or changing the permissions of a directory affect that directory and all its contents (sub-directories are treated recursively), and must occur atomically. Similarly, moving or copying a file from a location inside one partition to a location inside another partition also must occur atomically and serializably. Second, metadata is repeatedly accessed throughout file system requests for verifying paths, checking permissions, and finding data relevant to a request, and cannot afford excessive delays for multi-node coordination.

An alternative approach is to store the metadata in a distributed database system that manages the partitioning of the metadata, and guarantees atomicity, isolation, and durability of all transactions. However, performance and efficiency can be a problem. When the file system logic is running outside of the database system, there are typically many round trips between the file system logic and database layer for each file system request. These round trips can add up to substantial increased latency, and reduced efficiency of system resources.

To enhance storage scalability and improves availability to handle the massive growth in data needs of an ever-growing number of applications, by treating metadata management similar to data management, we built FileScale, an HDFS-based file system that replaces metadata management in HDFS with a three-tiered distributed architecture that incorporates a high throughput, distributed main-memory database system at the lowest layer, along with distributed caching and routing functionality above it, so that most requests can be served with asynchronous, batched interactions with the database layer. This architecture enables FileScale to be a simple upgrade over existing HDFS implementations in which all interfaces — both internally and externally — remain the same, and the performance on a single node is nearly identical to the original HDFS implementation. However, as the metadata scales, the architecture transparently partitions the metadata over a shared-nothing cluster of nodes, achieving linear scalability relative to performance on a single node.

1.3 Flock

Many high-volume data sources, such as sensor measurements, machine logs, user interactions on a website or mobile application, and the Internet of Things, operate in real time. Stream processing systems are critical to providing the freshest possible data and driving organizations to make faster and better automated decisions. These jobs show high variability and unpredictability, up to an order of magnitude more than the average load [5, 6]. This, along with the

broad variety of user SLOs, makes statically configuring and tuning streaming systems extremely difficult. Furthermore, traditional server-centric deployments use clusters provisioned with a fixed pool of storage and compute resources to execute these jobs, it can frequently suffer from resource under- or over-provisioning, leading to resource wastage or performance degradation, respectively.

Serverless platforms [7, 8, 9] fulfill the promise of transparent resource elasticity in the cloud [10, 11, 12]. Under the Function as a Service (FaaS) serverless model, developers decompose their applications into short-lived cloud functions. The ease of programming, fast elasticity, and fine-grained pricing in FaaS platforms allow for fine-grained scaling of resources to meet spiky demand, making them an appealing solution for streaming processing. The FaaS model provides more fine-grained elasticity with sub-second start-up times that can dynamically match the per-query basis with continuous scaling. Further, its billing methods are more fine-grained with millisecond granularity. Therefore a FaaS-based service is low cost to operate under low demand and can scale automatically to a high load at a proportional cost.

To explore the promise of function services for stream processing, we built Flock, a cloud-native streaming query engine that runs on FaaS platforms. Existing approaches [13, 14, 15, 16] take advantage of the on-demand elasticity of cloud object storage services, such as Amazon S3 [17] to shuffle data, which increases the performance cost and compromises the advantages of a serverless system. Instead, Flock passes data through the invocation's payload between cloud functions. This removes the need to read and write data from an external store

service. Payload invocation also eliminates the requirement for a query coordinator from the data architecture since Flock does not leverage any external storage service as a communication medium between functions. Flock supports the vectorized processing on ARM processors, which brings 20% speedup and reduce costs by more than 30% on x86. Flock is thus the first streaming query engine, to the best of our knowledge, to support standardized abstractions, SQL and Dataframe API, on cloud function services, allowing users and engineers to avoid the time-consuming process of manually translating SQL into cloud workflows on heterogeneous hardware.

1.4 BullFrog

Software as a Service, or SaaS, is a method of software product delivery to end-users over the internet and via pay-as-you-go pricing in which the software is centrally hosted and managed by the cloud service provider. Continuous Deployment (CD) in SaaS, an aspect of DevOps, is the increasingly popular practice of frequent, automated deployment of software changes. To realize the benefits of CD, it must be straightforward to deploy updates to both front-end code and the database, even when the database's schema has changed. Unfortunately, this is where current practices run into difficulty. Schema changes occurred roughly once per week in a dozen open-source applications [18]. Application developers should be free to change the code and database schema as they see fit, without concern for the complexities of deploying those changes later. Since downtime

frequently is a concern, and with multiple updates happening per day, the simple shutdown-and-restart approach is unacceptable.

We built BullFrog [19], a PostgreSQL extension that is the first system to use lazy schema migration to support single-step, online schema evolution without downtime, which achieves efficient, exactly-once physical migration of data under contention. When a schema migration is submitted, BullFrog initiates a logical switch to the new schema, but physically migrates affected data lazily, as it is accessed by incoming transactions. BullFrog's internal concurrency control algorithms and data structures enable concurrent processing of schema migration operations with post-migration transactions, while ensuring exactly-once migration of all old data into the physical layout required by the new schema. BullFrog can achieve zero-downtime migration to non-trivial new schemas with near-invisible impact on transaction throughput and latency.

1.5 Outline

The remaining of the thesis is organized as follows. In Chapter 2, we present FileScale, a three-tier architecture that incorporates a distributed database system as part of a comprehensive approach to metadata management in distributed file systems. In Chapter 3, we propose Flock, a cloud-native streaming query engine on cloud function services. In Chapter 4, we describe BullFrog, a relational DBMS that supports single-step schema migrations — even those that are backwards incompatible — without downtime, and without need for advanced warning. We

conclude this dissertation and discuss the future work directions in Chapter 5.

Chapter 2: FileScale: Fast and Elastic Metadata Management for Distributed File Systems

Recent work has shown that distributed database systems are a promising solution for scaling metadata management in scalable file systems. This work has shown that systems that store metadata on a single machine, or over a shared-disk abstraction, struggle to scale performance to deployments including billions of files. In contrast, leveraging a scalable, shared-nothing, distributed system for metadata storage can achieve much higher levels of scalability, without giving up high availability guarantees. However, for low-scale deployments – where metadata can fit in memory on a single machine – these systems that store metadata in a distributed database typically perform an order of magnitude worse than systems that store metadata in memory on a single machine. This has limited the impact of these distributed database approaches, since they are only currently applicable to file systems of extreme scale.

To address the challenge, we built FileScale, a three-tier architecture that incorporates a distributed database system as part of a comprehensive approach to metadata management in distributed file systems. In contrast to previous approaches, the architecture described in the chapter performs comparably to the

single-machine architecture like HDFS at a small scale, while enabling linear scalability as the file system metadata increases.

2.1 Introduction

As the data stored by organizations rapidly expands, both the structured metadata and unstructured byte contents of the files managed within file systems scale commensurately. In general, it is easier to scale the unstructured data than the structured data, since there is no requirement to perform atomic transactions that update the unstructured bits across multiple files. Therefore, unstructured data can simply be placed in blocks that are partitioned across a shared-nothing cluster of nodes (machines), and all operations on the unstructured data are straightforward to parallelize across this cluster, with little-to-no coordination across nodes except for replication.

However, scaling the structured data is more challenging for several reasons: First, there is a requirement for atomic, isolated, and durable transactions that may access data in multiple partitions. For example, recursively deleting or changing the permissions of a directory affect that directory and all its contents (sub-directories are treated recursively), and must occur atomically. Similarly, moving or copying a file from a location inside one partition to a location inside another partition also must occur atomically and serializably. Second, metadata is repeatedly accessed throughout file system requests for verifying paths, checking permissions, and finding data relevant to a request, and cannot afford excessive

delays for multi-node coordination.

The first generation of scalable file systems, such as GFS, HDFS, Lustre, Ursa Minor, Farsite, and XtreemFS [20, 21, 22, 23, 24, 25], focused on scaling the unstructured data linearly, but stored metadata in memory on a single machine. Despite the lack of scalability of metadata management, they managed to scale to petabytes of data by using block sizes on the order of megabytes or gigabytes, and limiting the number of unique files and directories under management, so that information about blocks, files and directories can fit in memory on the metadata node (which was typically provisioned with copious amounts of memory). These restrictions typically present no problems for data processing and large scale analysis workloads, which usually involve large scans and prefer large block sizes anyway. However, these restrictions are problematic for workloads that access data in smaller quantities. In addition, even large scale analysis workloads that use large blocks sizes are reaching the metadata limits of existing scalable file systems with increasing frequency.

Furthermore, using a single machine [26] to handle all metadata requests can become a performance bottleneck when it is overwhelmed by many concurrent client requests, along with processing heartbeats from the increasingly large numbers of block-store servers in the system. Furthermore, it becomes a single point of failure unless another machine that has identical provisions of copious memory and processing capability runs alongside it, ready to take over at any moment. Therefore, solutions that remove the memory limitations by incorporating fast external storage attached to the metadata node (e.g. [27, 28, 29, 30, 31, 32, 33,

34, 35]) will not be sufficient in the long run.

One approach to scaling metadata is to partition it, but restrict atomicity and isolation guarantees to only those requests that can be processed by a single partition. This is the approach taken by HDFS's federation option [36, 37] where the file system namespace is statically partitioned across completely independent "NameNode" servers that store disjoint partitions of file system metadata, with optional client-side routing tables [38] or a routing layer [39, 40, 41, 42] that direct metadata requests to the correct NameNode. Nonetheless, preventing multi-partition requests limits the general applicability of these approaches, and reduces the functionality of the file system.

An alternative approach is to store the metadata in a distributed database system that manages the partitioning of the metadata, and guarantees atomicity, isolation, and durability of all transactions — even those that span partitions [43, 44, 45][1]. These approaches have demonstrated that scalable database systems can successfully scale all aspects of file system metadata management. However, performance and efficiency can be a problem. When the file system logic is running outside of the database system, there are typically many round trips between the file system logic and database layer for each file system request. These round trips can add up to substantial increased latency, and reduced efficiency of system resources. For example, the HopsFS paper reported that it took 3 NameNodes and 2 database servers to match the throughput that the single active HDFS NameNode

[1]Although Colossus[46] and Giraffa [47] use scalable data stores (BigTable [48] and HBase [49]), they do not support multi-partition requests because they lack strongly consistent distributed transactions.

is able to achieve [44]. On the other hand, building the file system logic into the database system requires rip-and-replace upgrades of existing file system technology and has yet to be shown to be a generally applicable approach.

In this chapter, we describe the design of FileScale, an HDFS-based file system that replaces metadata management in HDFS with a three-tiered distributed architecture that incorporates a high throughput, distributed main-memory database system at the lowest layer, along with distributed caching and routing functionality above it, so that most requests can be served with asynchronous, batched interactions with the database layer. This architecture enables FileScale to be a simple upgrade over existing HDFS implementations in which all interfaces — both internally and externally — remain the same, and the performance on a single node is nearly identical to the original HDFS implementation. However, as the metadata scales, the architecture transparently partitions the metadata over a shared-nothing cluster of nodes, achieving linear scalability relative to performance on a single node. Experiments that compare FileScale to a recently published state-of-the-art file-system-over-database-system implementation show that FileScale is orders of magnitude more efficient.

2.2 HDFS Background

HDFS is perhaps the most widely deployed distributed file system today for machine learning and data analytics [50, 51, 52]. It uses a leader/follower architecture in which a NameNode manages all file system metadata and regulates

data access on behalf of clients. Files are split into one or more blocks, and these blocks are replicated across a set of DataNodes in a shared-nothing architecture. NameNodes execute metadata operations over the set of *inodes* accessed by a request. Each inode stores information about a file or directory. For read requests, it returns the list of blocks (and their DataNode location) that are accessed by the request. DataNodes interact directly with clients, serving read and write requests. They also perform block creation, deletion, and replication when instructed to do so by the NameNode.

NameNode durability is implemented via a write-ahead log called the `EditLog`. Recovery is performed by loading a checkpoint called a `FSImage` and then replaying the `EditLog` over this image file. This process can be time consuming. Therefore, for improved high availability, HDFS allows for the deployment of a hot-standby that continuously, asynchronously, keeps the `FSImage` merged with the `EditLog`, so that it can take over with only minor delay when the primary NameNode crashes or temporarily goes down.

2.3 System Architecture

FileScale is designed to serve as a drop-in replacement for HDFS, maintaining an identical API at the client side, and intercepting communication with the HDFS NameNode and redirecting it across FileScale's distributed NameNode implementation. FileScale replaces HDFS's single NameNode with a three-tiered architecture that collectively provide a more scalable implementation of the iden-

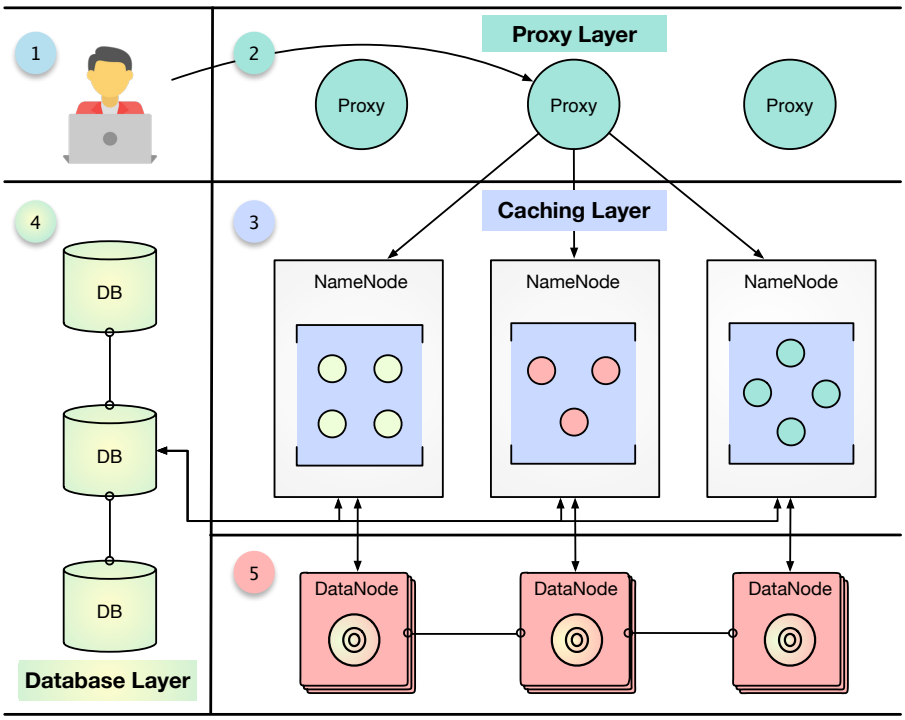


Figure 2.1: System Architecture of FileScale. Inodes are small circles that sit in the NameNode's memory.

tical NameNode functionality. These three tiers implement routing, caching, and stable storage of file system metadata.

The high level architectural design of FileScale is illustrated in Figure 2.1. When a client makes a request to FileScale, a proxy server receives the request and routes it to a NameNode based on the file path of that request. The NameNode functions in FileScale as a cache of a subset of metadata. If the metadata relevant to the request is currently in the cache of the NameNode that receives it, it can respond immediately. Otherwise, either the relevant data is brought into cache, or this request is forwarded and processed as a transaction in the database layer. The results of the request are then returned to the client, which typically contain locations of DataNodes where the raw data is stored. The DataNode code

in FileScale is identical to the DataNode code in HDFS. The following sections 2.4, 2.5 and 2.6 provide more detail on each layer of FileScale's architecture.

2.4 Database Layer

FileScale stores all file system metadata in a distributed database system that is partitioned and replicated across a shared-nothing cluster. FileScale uses a modular architecture such that any ACID-compliant SQL database system could be used. However, some functionality is implemented within stored procedures (see Section 2.4.2) so some new code is necessary to add support for a new system. The codebase currently supports VoltDB [53] and Apache Ignite [54].

2.4.1 Data Model

File systems typically store metadata as a tree of inodes with a root corresponding to the root directory, and children corresponding to directories and files located in the parent directory. Files are always leaves of this tree (i.e. they have no children) and they point to data block references from which the data associated with this file can be read. In HDFS this entire tree is stored in the memory of the NameNode.

Since FileScale uses a relational database system, the tree data structure must be transformed into a relational schema. The relational schema in FileScale contains 14 tables, consisting of two tables associated with the two main entities: `inodes` and `datablocks`, along with several relationship tables such as mappings

from inodes to blocks, and blocks to storage locations (DataNodes). Table 2.1 shows a simplified version of the schema used in FileScale. The *pid* and *pname* attributes of the `inodes` table enables the reconstruction of the parent-child relationships from the original tree.

We partition the `inodes` table across the database system cluster via the attribute *pname*[2]. All tables that have 1:n relationships with the `inodes` table, such as the `datablocks` table, are partitioned based on their association with the `inodes` table, in order to maximize locality. The remaining (small) tables are replicated across the cluster.

[2]The full file path (`pname`, `name`) is the primary key of the `inode` table.

inode2block

block-id	id	index
1073741825	16386	0
1073741826	16386	1
1073741827	16386	2
1073741828	16388	0
1073741829	16389	0

datablocks

block-id	kbytes	stamp	replica
1073741825	131072	1001	1
1073741826	131072	1002	1
1073741827	45056	1003	1
1073741828	6.6	1004	1
1073741829	1628.2	1005	1

block2storage

block-id	idx	storage-id
1073741825	0	DS-e3d5de23
1073741826	0	DS-e3d5de23
1073741827	0	DS-08989547
1073741828	0	DS-dc8aa54e
1073741829	0	DS-dc8aa54e

inodes

id	pid	pname	name	access-time	update-time	header	permission
16385	0	null	/	0	1545261571024	0	1099511693805
16386	16385	/	event_data	1545267685278	1545264231090	281474976710672	1099511693823
16387	16385	/	dnn_model	0	1545267685104	0	1099511693805
16388	16386	/dnn_model	graph.ckpt.pbtxt	1545267685125	1545267685125	281474976710672	1099511693823
16389	16386	/dnn_model	model.ckpt.data0	1545267685224	1545267685224	281474976710672	1099511693823

Table 2.1: Data model in FileScale.

2.4.2 Transaction Processing

After storing metadata in the database, metadata operations are performed as atomic transactions over the database system. The metadata component of some file system commands can be transformed into a series of simple INSERT, UPDATE, or SELECT statements over the database system. However, other commands require computation between these statements such that there is an interleaving of statement execution and application code. Either way, FileScale implements all interactions with the database system via atomic, pre-compiled stored procedures in order to leverage its more advanced knowledge of data partitioning details.

2.5 Caching Layer

Each metadata operation in a file system must resolve path components recursively to validate the entire path and check user permissions and quota configuration. The multiple round trips back and forth to the database system required during the recursive path resolution process can result in substantial latency — even when the underlying database system stores all data in main memory. FileScale therefore introduces a caching layer in each NameNode's memory that enables a copy of a set of metadata objects (such as inodes) to be stored in local memory, which can be accessed directly by metadata operations and thereby avoid communication with the database system upon a cache hit. Updates to metadata stored inside a FileScale cache are not propagated to the underlying

database system until an event occurs that requires propagation, such as an expiration, periodic flush, or distributed transaction. Thus, the database layer lags behind the cache layer, and up-to-date access to records in the database layer may require synchronization activities with the cache layer prior to serving those accessed records.

2.5.1 Object Cache

The mappings of files to blocks and blocks to DataNodes in HDFS's namespace are implemented as a light-weight hash table in HDFS whose primary goal is to optimize memory usage within the NameNode [55]. This enables the entire metadata to fit in memory and supports high throughput concurrent client request processing. In contrast, in FileScale, these mappings are stored in an in-memory object cache. This design enables FileScale to avoid the need to assume that all data fits in main memory, and allows the caching layer to function like a cache in which individual objects are continuously added and removed from memory.

However, an important advantage of HDFS's assumption that all metadata fits in memory on the NameNode is that this metadata can be given a permanent location in memory that can be directly referenced by all other metadata that refer to it. For example, a directory needs to refer to all of its immediate children: the files and directories stored inside of it. With HDFS's design, the children of an inode can be stored as an in-memory list of direct pointers to the location of these

children inodes in memory. In order to resolve a complete path, HDFS simply needs to start at the root, and follow the series of direct pointers from root to the next child, and from that child to the next child, etc.

In contrast, in FileScale's cache-based design, metadata cannot live in a permanent location in memory, since each cached object may be evicted according to the cache eviction algorithm. Therefore, each object is given a globally unique identifier, and references to objects, such as the children of a directory, are done via specifying the identifier instead of via a direct pointer. A separate lookup must occur to find the current location of the identified object in memory.

Although the extra lookups can cause increased latency relative to a direct pointer approach in some cases, it enables a performance optimization that allows FileScale to validate paths quicker than the direct pointer approach. Using the direct pointer approach, a search must occur at each level of the path being validated: specifically the child specified by the path must be located amongst all of the other children of the same directory. This is implemented in HDFS via a binary search within the list of children of a directory. FileScale eliminates the need for this search at each level since the reference to the child is derived directly from the path name of the child (see example below). For directories with many children, the hash lookup by path is cheaper than a binary search within the list of pointers to children. Furthermore, each element of a path can be searched for independently, and in parallel, instead of sequentially having to traverse from parent to child.

An example path resolution (`/tmp/logs/data`) is shown in Figure 2.2. In

the forward resolution, the root node has no parent, so it is searched for as </>. The root node (16385) is retrieved to verify its existence and check its permissions. The next element in the path is then searched for: </tmp> which returns the inode with ID 16386. </tmp/logs> is then searched which returns the bucket with ID 16387. Finally, </tmp/logs/data> is searched which returns the bucket associated with the end of the path. The steps outlined in this example do not need to be sequential. The nodes </>, </tmp>, </tmp/logs>, and </tmp/logs/data> can all be searched for in parallel since these four lookup keys are all directly derived from the full path (/tmp/logs/data). Thus, in practice, there is no distinction between forward resolution and backward resolution of file paths. The performance advantages of this approach relative to performing a search at each level of the path will be explored in Section 2.7.2.3.

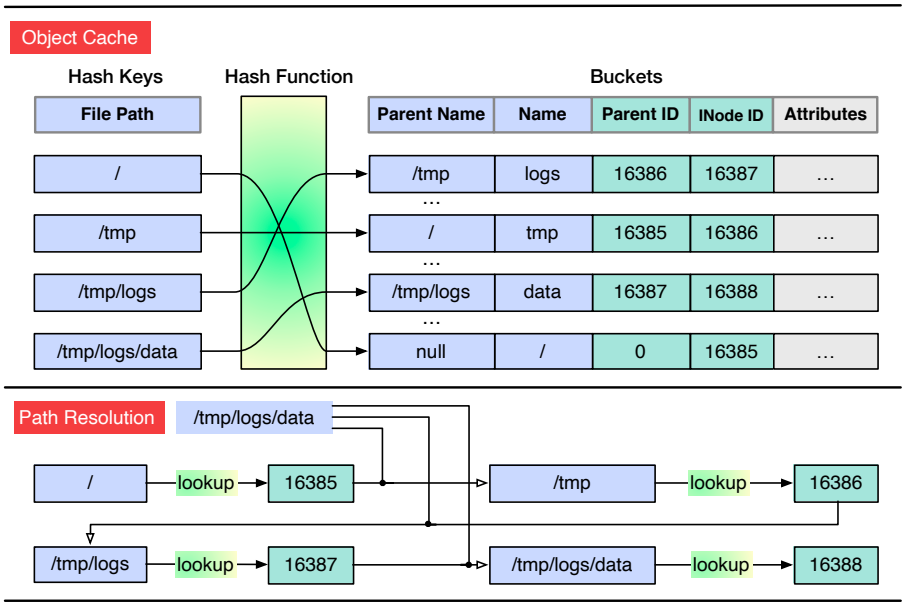


Figure 2.2: Path resolution.

2.5.2 Durability

Since updates to data in cache are not immediately propagated to the database, the database log is not sufficient to guarantee system-wide durability. Therefore, the cache layer implements a write-ahead logging mechanism based on an extension of HDFS's `EditLog`, and the database log is used only for transactions that are not performed in cache. Each NameNode logs all modifications it makes to a separate log file stored remotely in a network file system[3]. Locks are not released until the logging service acknowledges the writes.

A periodic process asynchronously flushes recent writes from the cache layer to the database layer, in batches. This limits the staleness of data in database layer. A background process in the database layer takes periodic durable checkpoints of a snapshot of transaction-consistent database system state. Recovery starts from the most recent checkpoint, and plays forward any log records found in the logging service that were not incorporated in the database state, which are merged with log records found in the database log.

Log records that are reflected in any database layer checkpoint can be safely removed from the logging service.

Figure 2.3 shows the workflow of file-create operation. When FileScale receives a request to create a file with ID = 7, the NameNode writes a log record to the remote server and creates an inode object in the cache. After the NameNode receives a success message from the logging service, it can make the inode visible

[3]HDFS similarly stores edit logs on Quorum Journal Machines [56] or NFS [57] for high availability.

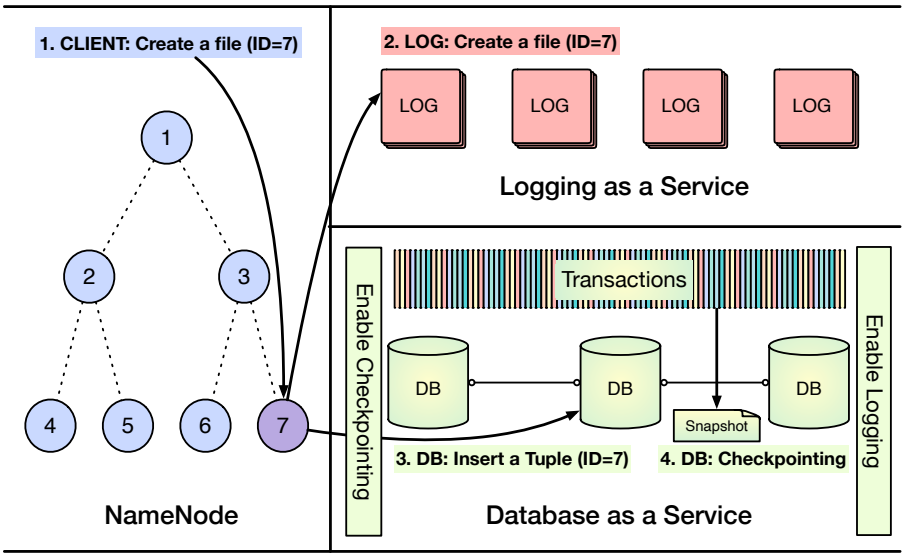


Figure 2.3: The workflow of file-create (metadata) operation.

to subsequent requests prior to flushing the write to the database layer. Eventually the write is flushed to the database layer and is incorporated into a database snapshot, after which the log record associated with that write can be safely truncated from the log.

2.6 Proxy Layer

FileScale horizontally scales the name service through the creation of multiple, independent NameNodes in the caching layer. Each NameNode manages a disjoint partition of the name space. However, the union of all the partitions need not cover the entire namespace. Requests over partitions of the namespace not covered by a NameNode are performed directly by the database layer. FileScale implements a proxy layer to route requests to the appropriate NameNodes that will process those requests. Unlike HDFS, FileScale supports multi-partition (multi-NameNode) transactions.

2.6.1 Request Routing

FileScale stores the namespace partitioning across NameNodes in a "mount table" stored in Zookeeper [58]. Specific file path prefixes are assigned to NameNodes. The assigned NameNode manages all metadata associated with all paths that begin with the specified prefix. Most of this metadata will be cached in memory at that NameNode, and the rest only accessible from the database layer. An example mapping of file paths to NameNodes is shown below.

```
   File Path      |       NameNode
------------------+-------------------------
 /nlps/train_data | hdfs://192.168.1.1:9000
 /tmp/common/data | hdfs://192.168.1.2:9000
 /tmp/common/logs | hdfs://192.168.1.3:9000
```

The mount table is updated when new NameNodes are added or removed from the cluster, or when partitions need to be combined or split for improved load balancing. In practice it is read far more frequently than it is updated. Therefore, routing paths can be cached at the individual servers of the proxy layer for improved performance. However, this results in the proxy layer occasionally routing a request to the wrong NameNode, and that NameNode must then forward the request to the correct one (see below).

FileScale supports two modes to route user requests: (1) proxy mode and (2) watch mode. In proxy mode, the proxy layer consists of multiple routers that use the same communication protocols as HDFS. The router acts as a middleware layer that includes an upstream manager that maintains communication sessions

for different clients, and intercepts client requests/responses to manipulate them as needed. The proxy layer can share hardware with the caching layer, such that there exists a router on each NameNode. When a client request is received by a router, the file paths associated with that request are extracted, and longest prefix matching is performed to locate the mount table entries relevant to that request. If all items accessed by the request are managed by a single NameNode, the request is forwarded there. Requests accessing data not associated with a NameNode are sent directly to the database. Multi-partition requests are sent to the database layer after a synchronization occurs between the cache layer and database layer (Section 2.6.2).

Watch mode works identically to proxy mode, except that the client watches ZooKeeper and caches the mount table at the client-side to save a network hop. The performance benefits of watch mode will be explored in Section 2.7.3.1.

Figure 2.4 shows an example in which a file-open request (open file `/a`) is routed to the appropriate NameNode. The two different sets of blue lines correspond to the proxy and watch modes described above.

In both the watch mode and proxy mode, mount table data is cached locally and a listener is registered in order to receive notifications when changes occur. On occasion, a name space partition may be moved from one NameNode to another, or an existing partition may be split or combined with a partition located on a different NameNode, temporarily rendering these caches stale, and causing misrouting of requests. Each NameNode maintains a recent-memory of paths that it

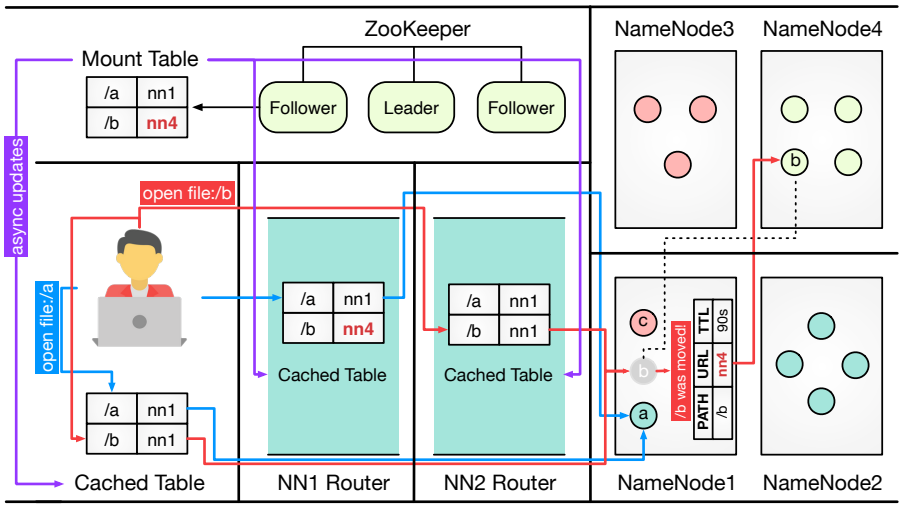


Figure 2.4: Request Routing in FileScale.

formally managed, but part or all of it was moved to a new NameNode[4]. This enables the NameNode to immediately forward requests that were misrouted to it based on stale information to the NameNode that took over the management of that partition of the namespace. This recent memory of moved paths is maintained with a short Time to Live (TTL) for each entry, since the cache of the mount table at each location is typically updated with short delay after it becomes stale. In the rare occasion where a NameNode receives a misrouted request for which it has no entry in its list of recent moves (because the TTL for that entry was too short and the entry already expired), the NameNode must look up the correct routing information in ZooKeeper to properly reroute the request.

In Figure 2.4, the two red lines sent from the client (open file /b) illustrate this process. The requests are forwarded to the wrong NameNode because of outdated routing information and are then forwarded to the correct location directly

[4]There is always a root path '/' in the mount table. Therefore adding a new path to the mount table is equivalent to splitting the root path into (1) the new path (2) everything else that was formally included in the original root path partition.

from the old NameNode.

2.6.2 Multi-partition requests

File systems that partition by path prefixes reduce the frequency of multi-partition transactions; however, they still occur. The main source of multi-partition requests are 'move' or 'copy' operations in which data from the source partition must be read (in the case of 'copy' operations) or removed (in the case of 'move' operations) and must also be inserted into the destination partition. Occasionally multi-partition requests are submitted outside the context of move or copy operations. For example, a recursive 'chmod' (change the file permissions) or 'rm' (delete) that starts high in the directory tree (close to the root) may span partitions.

In FileScale, all multi-partition requests are performed by the database system after all data accessed by the transaction are removed from cache (dirty data is written to the database prior to removal) and prevented from being brought into cache while the transaction is ongoing.

Figure 2.5 depicts the control flow between the NameNodes and associated services when a directory move operation spans multiple partitions. In this example, the directory with inode ID of 3 (along with its children) is being moved from a source partition to a destination partition managed by a different NameNode. (1) The source NameNode writes back all relevant dirty inodes in batches and removes the subtree from the cache layer. (2) The database layer is updated

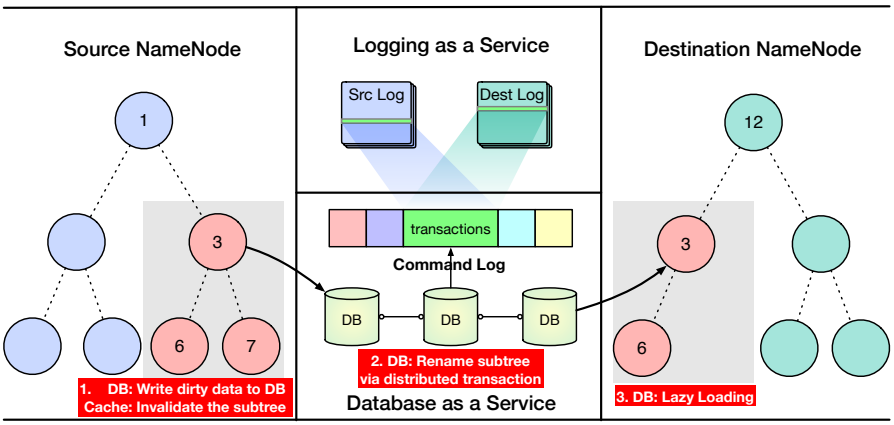


Figure 2.5: Move a folder across NameNodes.

synchronously via a (distributed) transaction that updates all affected inodes' names and their parent names. The precise implementation of the transaction depends on the underlying system, but can often be implemented via the SQL `LIKE` or `STARTS WITH` clause.

(3) The destination NameNode can choose to load the entire new subtree or lazily load it as needed.

FileScale's cache layer log appends the offsets of the database log of the multi-partition transaction after it completes. The helps FileScale properly interleave log records from the cache layer and the database layer during recovery.

2.7 Performance Evaluation

The implementation of FileScale directly inside the HDFS codebase was a large engineering effort and produced a total of 40k lines of code in HDFS 3.3.0. This effort allows for direct comparison of the metadata scalability and performance of FileScale with standard HDFS along with a state-of-the-art HDFS alter-

native that also stores data in a distributed database system (HopsFS) [44].

We initially use VoltDB [53] for FileScale's database layer. VoltDB is an in-memory DBMS that implements durability via a combination of asynchronous checkpointing and synchronous command logging that can be deterministically replayed to arrive at the state prior to a crash. In Section 2.7.5 we investigate the performance consequence of replacing VoltDB with Apache Ignite [54].

2.7.1 Experimental Setup

Previous attempts to scale metadata management within HDFS have succeeded in scaling file system throughput far beyond what a single HDFS NameNode is able to achieve. However, this comes at a cost of efficiency. For example, the HopsFS paper reported that it took 3 NameNodes and 2 database servers to match the throughput that the single active NameNode is able to achieve [44] (see Figure 6 from that paper). A major goal of FileScale's architecture is to enable file system scalability with a higher amount of efficiency, so that it can be used from the early stages of an application up through the later stages as the application scales over time.

To that end, our experiments focus on **both small and large deployments**, ranging from running on a single server to large clusters of servers running on Amazon Web Services (AWS) EC2 instances. All experiments are run on EC2 t3a.2xlarge[5] instances for NameNodes and database servers. Each EC2 instance

[5]Each instance contains 32 GiB of memory, 8 VCPUs feature the 2.5 GHz AMD EPYC 7000 series processors and 5 Gbps of network burst bandwidth.

attached a EBS volume optimized for transactional workloads, and the volume is a 128 GiB of Provisioned IOPS (io1) SSD that can provision up to 64000 IOPS. Optimal NameNode heap size depends on many factors, such as the number of files, the number of blocks, and the load on the system, and generally requires tweaking since each workload has a unique byte-distribution profile. To reach the NameNode memory bottleneck quickly for our experiments, we use 16 GB for heap memory and garbage collection.

We use the NNThroughoutBenchmark [59] to generate test workloads. It's a NameNode throughput benchmark, which runs a series of client threads against a NameNode. However, the benchmark code out of the box runs on a single node without end-to-end network latency, so we extended the client workload generation in the benchmark codebase to run in the large-scale environments required for our analysis in this section.

2.7.2 Single-node Experiments

We start by comparing FileScale with HDFS version 3.3.0 and HopsFS on a single AWS EC2 instance. All systems use a single NameNode, and the database servers used by FileScale (VoltDB) and HopsFS (NDB) run on the same machine as the NameNode.

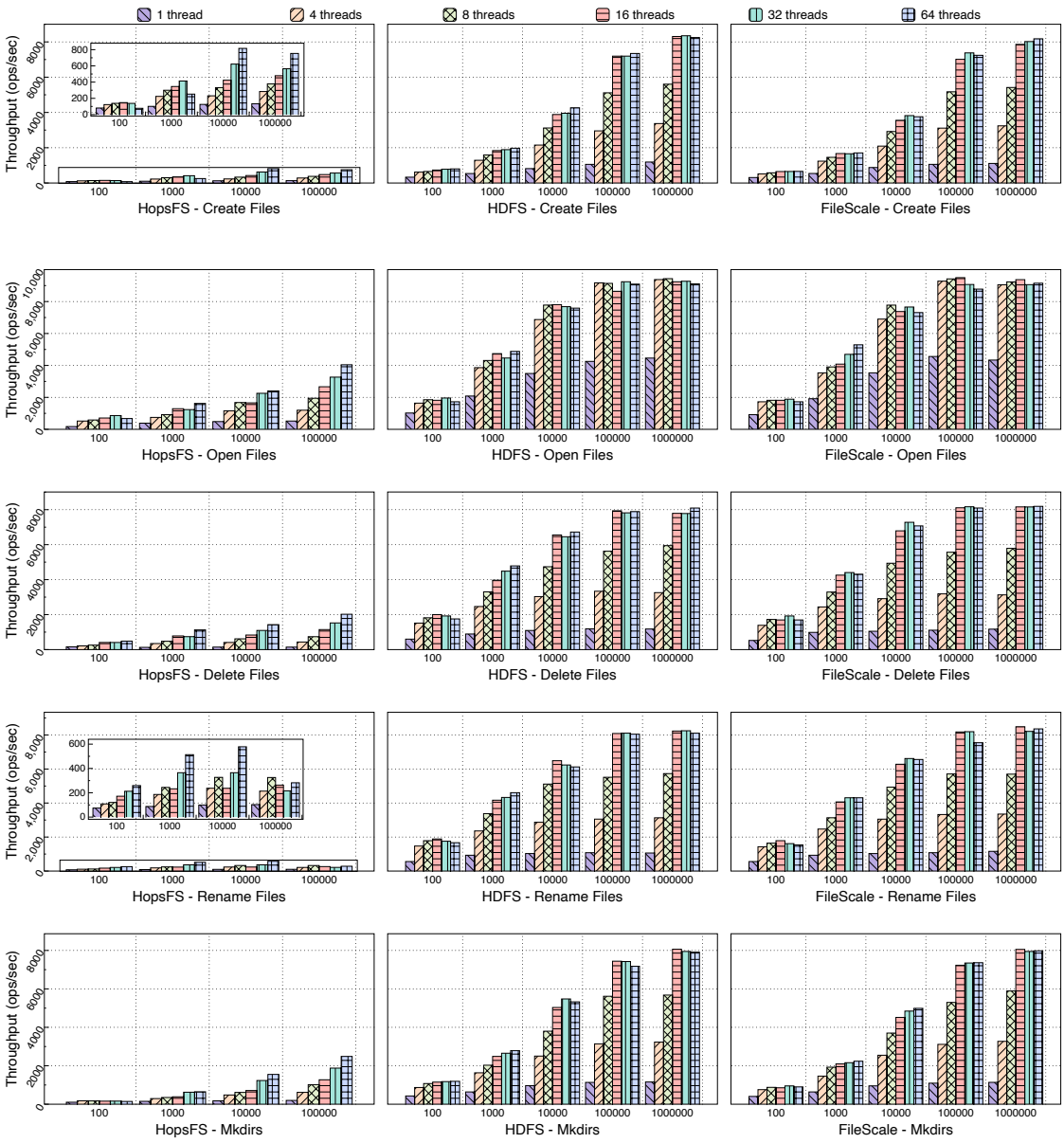


Figure 2.6: The throughput of basic operations including *create*, *open*, *delete*, *rename* and *mkdirs* on a EC2 instace—t3a.2xlarge.

2.7.2.1 Basic Operations

Figure 2.6 shows the throughput of directory create and file create, open, delete, and rename operations while varying the total number of each of these operations run (i.e., the number of files created, opened, etc.), and the number of

client threads.

For all types of operations, the performance of FileScale and HDFS is similar. This is because both FileScale and HDFS store all metadata in memory when it fits on a single node and the performance of their respective in-memory data structures are similar. HopsFS ran out of memory after operations on over 100,000 files (a standard HopsFS deployment would divide the metadata and workload across many machines in order to avoid running out of memory). For operations on 100,000 files and fewer, the throughput of HopsFS was approximately one tenth of HDFS and FileScale when creating files and renaming files, and one fifth when opening files, deleting files, and creating directories. These results are roughly consistent with the numbers reported in the original HopsFS paper where it was reported that it took a total of 5 servers—3 NameNodes and 2 database servers—to match the throughput that the single active NameNode [44]. The main reason for the difference in performance between HopsFS and FileScale is that FileScale is able to avoid a round trip to the database system on the critical path during request processing when all data fits in cache. FileScale persists all updates to its write-ahead log (which has similar performance as writes to HDFS's write-ahead log) for durability and recovery. This enables FileScale to avoid being forced to rely on the database system for durability, and therefore it can propagate updates to the database system asynchronously, in batches. In contrast, every HopsFS request requires at least one synchronous round trip to the database system.

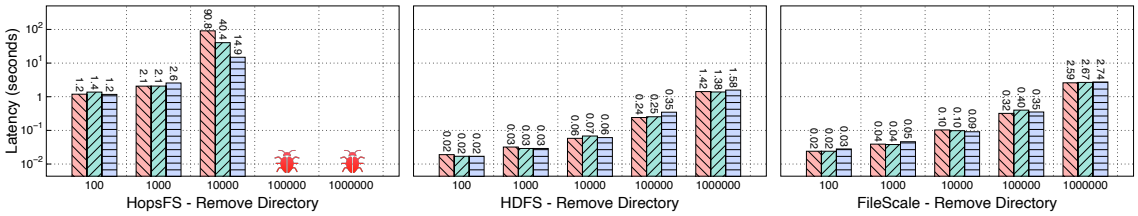


Figure 2.7: Recursive delete all files under the root directory.

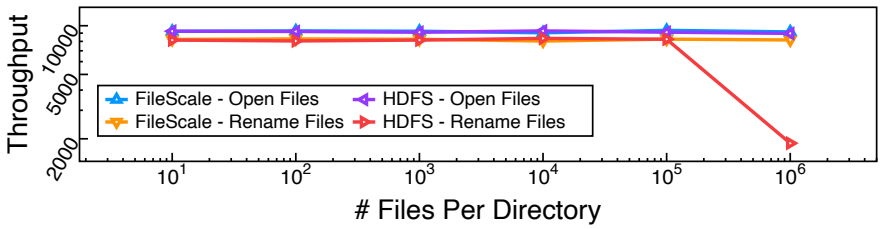
2.7.2.2 Recursive Delete Operations

FileScale caches inodes in memory in a format that enables a one-to-one mapping with relational tuples in the database system. In contrast, HDFS's primary storage of file system metadata is in memory, and parent nodes can store direct, in-memory pointers to children nodes. As was explained in Section 2.5, this makes operations that traverse the directory structure, such as recursive file system operations, slower in FileScale relative to HDFS. To understand this tradeoff in more detail, we ran an experiment in which we measured the latency of performing a recursive delete operation, while varying the number of files and the depth of the tree being deleted. The results are shown in Figure 2.7.

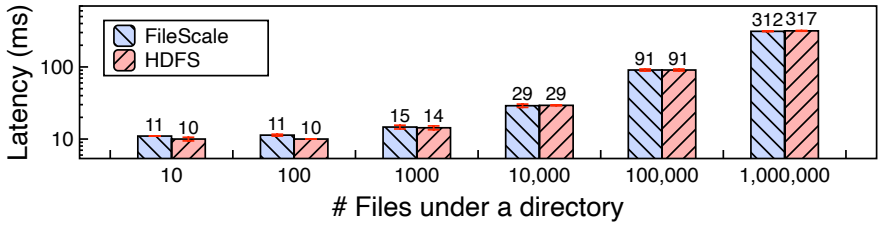
The results show that the primary bottleneck is the process of deleting each individual file. The latency of all systems therefore increased as the number of files being deleted increased. To delete a file, HDFS needs to remove the file in memory (along with writing a log record to stable storage), while FileScale invalidates nodes in its cache, writes a log record, and asynchronously deletes related tuples in the database system. Since the latency of the individual deletes was the bottleneck, the latency numbers were only slightly impacted when height of the

tree changed (when keeping the number of deleted files constant). Nonetheless, as expected, the overall latency of HDFS was slightly faster than FileScale. This is because FileScale's caching layer must be robust to situations in which child inodes are removed from the cache. Therefore, FileScale uses identifiers instead of direct pointers to children, and require a hashmap lookup the current location in memory of a particular ID.

The HopsFS codebase runs into problems in which transactions continuously timeout when we ran this experiment at more than or equal to 100,000 files (that appear to be caused by deadlocks). For the smaller experiments, we found that the latency of HopsFS are between one and two orders of magnitude worse than HDFS and FileScale (the figure uses a log scale y-axis). The relative performance between HopsFS and HDFS is consistent with the results from Section 7.4.1 of the HopsFS paper [44] where it is explained that HopsFS performs poorly on these types of workloads because they are executed in many separate small transaction batches. Surprisingly, the performance of HopsFS *improved* when the depth of the tree being deleted increased. This is because deleting directories that contained a large number of files resulted in increased lock contention for the directory lock. By increasing the depth of the tree being deleted, there were fewer files per directory to delete, which reduced lock contention.



Total throughput varying the depth of 10^6 files.



The latency of `ls` operations.

Figure 2.8: Large directory experiment.

2.7.2.3 Large Directories

Figure 2.8(a) shows the throughput across 64 threads performing file open and rename operations within a directory that varies in size from 10 to 1,000,000 files stored within it.

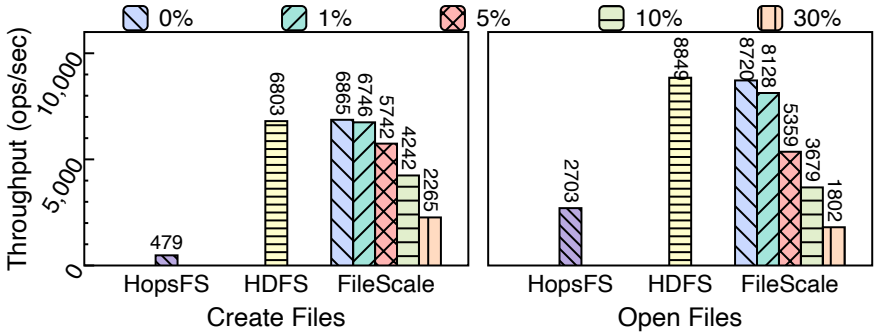
As can been seen, for most directory sizes, the performance of HDFS and FileScale are similar. However, in extreme cases, when directories contain a million files, the performance of the rename operation in HDFS drops substantially. This is because the list of children of a directory are stored as an `ArrayList`, and renaming files requires deleting elements from this list and reinserting them in order to keep the list in sorted order. Over time, these deletes and insertions require the entire `ArrayList` to be copied to a new location to improve the efficiency of how it is laid out in memory. However, copying a list that contains 1,000,000 causes a noticeable increase in latency, which drags down system throughput.

In contrast, FileScale does not require the list of children be kept in sorted order, since path validation does not require a binary search at each level (as we explained in Section 2.5). Instead, they are stored in a standard linked list that does not incur resize costs as the list grows in size. Figure 2.8(b) shows the latency of performing `ls` operations within a directory. Even though FileScale requires additional retrievals from object cache, FileScale and HDFS have similar latency since the serialization step of providing the output is the performance bottleneck.

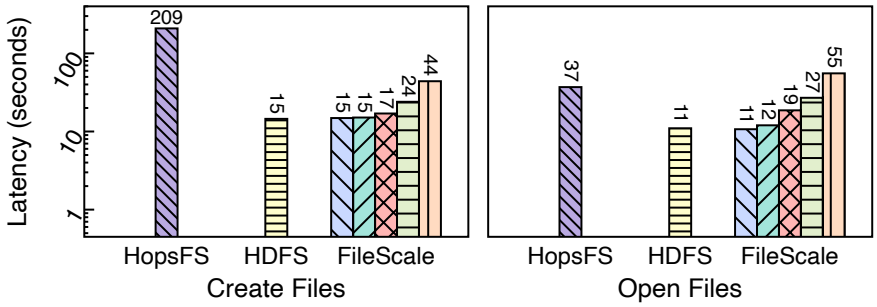
2.7.2.4 Cache Misses

One important difference in the design between FileScale and HDFS is that HDFS requires that all inodes fit in memory, whereas FileScale treats memory as a cache of the stable state of the inodes in the underlying database system. This enables FileScale to stay online even for deployments where there is not enough aggregate memory across the nodes in the deployment to store all inodes in memory. To understand the extent of the performance drop at reduced memory deployments, we ran an experiment in which we increase the cache miss rate of FileScale from 0% to 30% and measure the throughput reduction and latency increase on file create and open operations. The results are shown in Figure 2.9, and the original results for HDFS and HopsFS (from the previous experiments) are shown for comparison. For each experiment, we used 16 threads to operate on 100,000 files concurrently.

The throughput of FileScale degrades gracefully as the cache miss rate in-



The throughput of creating and opening files.



The latency of creating and opening files.

Figure 2.9: Cache miss penalty.

creases, with the throughput of create file operations at 30% cache misses approximately 3 times smaller than the throughput at 0% cache misses. Similarly, the latency of FileScale increases gracefully as the cache miss rate increases, with the latency of create file operations at 30% cache misses approximately 3 times longer than the latency at 0% cache misses. The performance of open file operations degrades more rapidly than for create operations (a factor of 5 drop instead of a factor of 3) because opening files can be served entirely from memory when data is in cache[6]. In contrast, creating files always has to push a log record to stable storage before the operation can commit regardless of whether the relevant directory data is already in cache, so the relative cost of a cache miss is smaller.

[6]In the supplemental material submitted with this paper, we show that switching the database system can reduce the size of this performance drop.

In practice, the number of cache misses in FileScale can be monitored and action taken to alleviate performance problems due to cache misses. Specifically, the proxy server in FileScale can leverage Hadoop's existing monitoring component (that collects various performance metrics) to immediately re-balance the mount table in FileScale's state store when performance decreases due to cache misses.

2.7.3 Multi-server Experiments

We next experiment with multi-server deployments in order to investigate the scalability of the different system architectures. We start with relatively small five-node deployments. HDFS does not support partitioning the same file system namespace across multiple NameNodes[7], but it can use the additional nodes to support HA (high availability). Therefore we set it up to use two NameNodes (in an active/standby configuration) and three JournalNodes. The journal nodes are used by HDFS to share logs between the active and standby NameNodes. When a NameNode writes a log record, it must be written to a majority of JournalNodes before it is considered durable.

FileScale and HopsFS use a similar configuration: two of the five nodes are used for NameNodes (but unlike HDFS, the metadata can be partitioned across them), and the remaining three nodes for the database system — VoltDB for FileScale and NDB for HopsFS. For HA, log records written by FileScale and

[7]HDFS does support partitioning metadata across NameNodes for different namespaces. Although being forced to partition the namespace significantly reduces the practical utility of HDFS "federation", we will investigate the performance of this alternative architecture in Section 2.7.3.1.

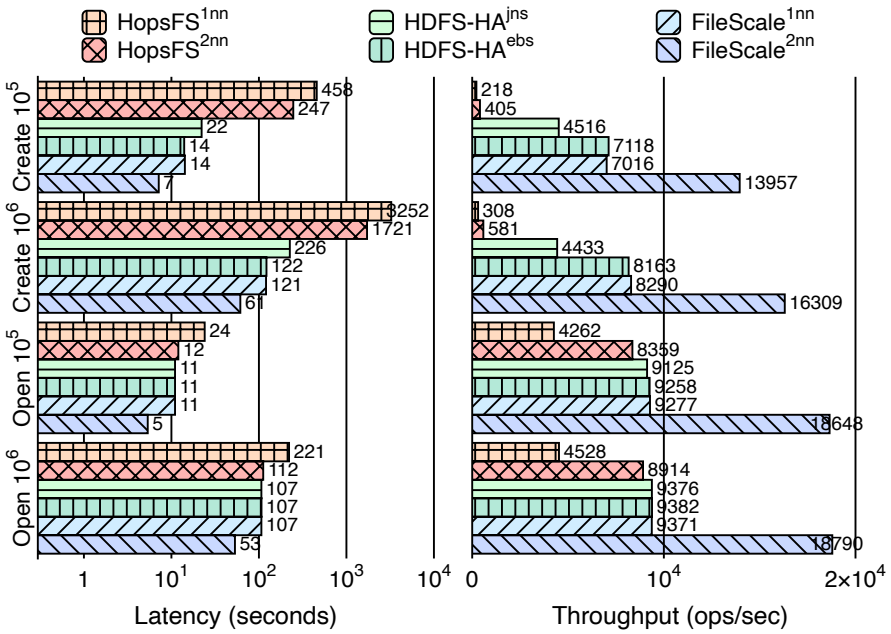


Figure 2.10: A five-node deployment.

HopsFS are written to EBS volumes so that they remain persistent beyond the life span of AWS EC2 instances (which use only ephemeral storage). For fairness, we also run a version of HDFS in which log files are written to EBS (which we call HDFS-HA^{ebs}) as an alternative to the version in which log files are written to journal nodes as described above (which we call HDFS-HA^{jns}).

Figure 2.10 shows the throughput and latency of file-create and file-open operations under this deployment. The performance of HopsFS is almost two orders of magnitude slower than FileScale for the file-create benchmark, so the figure is uses a log scale. This difference is consistent with the results presented in Figure 2.6 that show a large efficiency gap between FileScale and HopsFS. However, consistent with the claims from the original paper, HopsFS approximately doubles its performance as the number of NameNodes doubles from one NameNode (HopsFS^{1nn}) to two NameNodes (HopsFS^{2nn}). However, FileScale's

performance also doubles, so the difference in performance between FileScale and HopsFS remains constant. Since opening files does not require writes to the database system, a disadvantage of HopsFS (that it must synchronously write data to the underlying database) is not present, and the performance gap between FileScale and HopsFS is more narrow for the 'file open' workload relative to the 'file create' workload.

Writing to EBS instead of the journal nodes significantly improves the performance of HDFS for the file create workload, but makes no difference for the file open workload which do not require log records to be written. This enables the performance of HDFS and FileScale to be equivalent when running on one NameNode as they were in Figure 2.6. However the performance of FileScale doubles when doubling the number of NameNodes since it can partition data across them, whereas HDFS does not partition data and performance remains constant when adding the additional NameNode.

2.7.3.1 Scalability

We next increase the scale of our experiment by varying the number of NameNodes from 1 to 32 while keeping the number of database nodes constant. The workload consists of 50% file-create operations and 50% file-open operations that are uniformly spread across the namespace. We run two distinct HDFS architectures. The first is the default HDFS architecture in which the entire file system belongs to a unified namespace, so that there are no restrictions in the file sys-

tem operations that can be run. However, this version must store all file system metadata on a single NameNode, as we described above. We also experiment with HDFS's router-based federation (RBF) capability, in which the namespace is partitioned, and the associated metadata for each partition can be managed by different NameNodes. Although the functionality of HDFS RBF is severely limited — for example, the distributed rename operations across partitions (which we experiment with in Section 2.7.3.2) cannot be supported — it can support the simple file open and create operations used for this benchmark, so we experiment with it in this section.

HopsFS ran into a bottleneck at the database layer at 16 NameNodes. Therefore we ran two versions of HopsFS — one with only three database nodes where the bottleneck is observed, and one with eight database nodes that avoids the bottleneck. FileScale did not experience the same bottleneck since it puts less pressure on the database layer by writing to the database in batches asynchronously instead of issuing synchronous writes for each new file created. Furthermore, HopsFS issues a batch query to the database layer at the beginning of every request in order to retrieve all the relevant inodes for the file path components of the request. This can be avoided in FileScale when the relevant data is in cache. Therefore, FileScale only requires 3 database nodes throughout this experiment.

The results of this experiment are presented in Figure 2.11. HopsFS-8 NDB, HDFS-RBF, and both versions of FileScale are able to scale linearly — as the number of NameNodes double, so too does the total system throughput. Therefore, the original relative differences in performance between HopsFS, HDFS, and

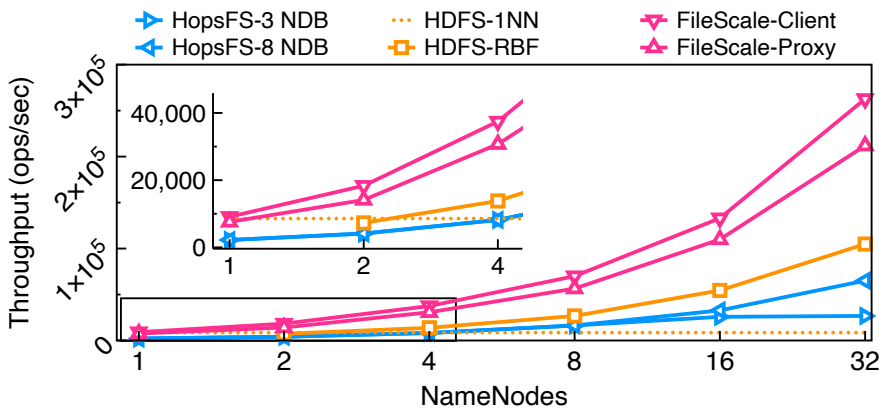


Figure 2.11: Throughput when scaling NameNodes.

FileScale observed when running on a single NameNode (see Figure 2.6) remain present as the system scales. However, HDFS-RBF has in effect half has many NameNodes as FileScale and HopsFS since HDFS requires one hot standby for every NameNode for high availability. As expected, HDFS-RBF outperforms HDFS's default implementation, since the default implementation cannot partition metadata across the additional NameNodes and thus cannot scale. Nonetheless, HDFS's default implementation (along with HopsFS and FileScale) is able support the full range of file system operations over all metadata, while HDFS-RBF must partition the namespace.

FileScale-Client corresponds to the watch mode configuration of FileScale described in Section 2.6, while FileScale-Proxy uses proxy mode. As expected, watch mode performs better since it is able to save a network hop during request processing, and avoid the overhead of processing and forwarding network messages at the proxy layer which shares physical hardware with the cache layer in the FileScale-Proxy deployment for this experiment.

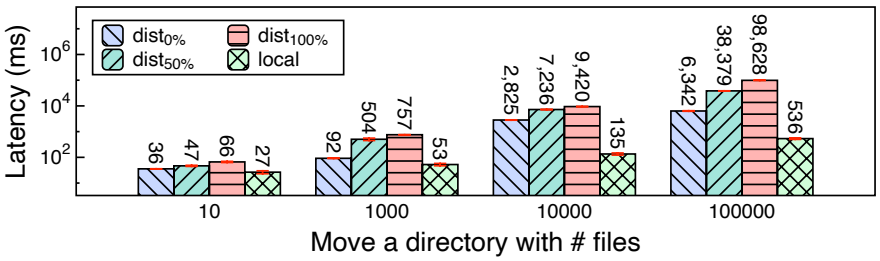
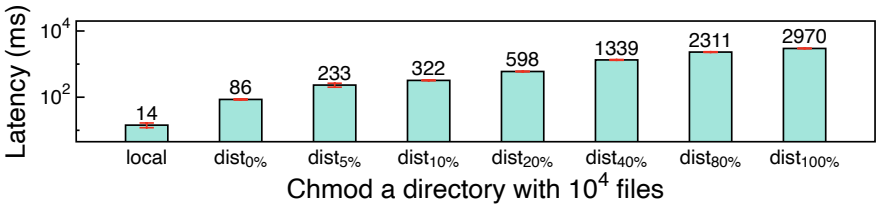


Figure 2.12: Local vs. distributed move operations.

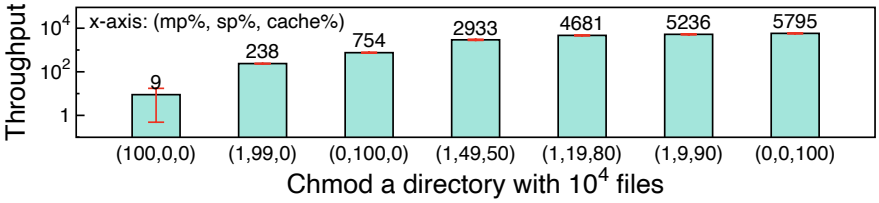
2.7.3.2 Multi-Partition Transactions

As explained in Section 2.6.2, FileScale performs multi-partition transactions at the database layer (after it is synchronized with the cache layer). To observe the performance impact of such transactions, we ran two experiments that involved multi-partition move and chmod operations.

Figure 2.12 shows the average latency of multi-partition move requests as the percentage of dirty data that must be written back to the database is varied. When the move operation is single-partition ("local"), the latency of the operation is limited by the time to generate and write the associated log records to the logging service. The more files being moved, the longer the latency. For multi-partition move requests, log records need to be written to two different log files (the log files associated with the source and destination NameNodes). Furthermore, the move operation requires updating the primary key (the full path of the file), which is an expensive operation in the database layer that partitions by the primary key and therefore must move data around when the primary key changes. Nonetheless, when significant amounts of data need to be written back to the database layer prior to the move operation, this write-back becomes the bot-



The latency of changing a directory's permission.



The total throughput of `chmod` operations.

Figure 2.13: Local vs. distributed chmod operations.

tleneck. The figure shows an order of magnitude difference when no data must be written back ($dist_{0\%}$) vs. all data must be written back ($dist_{100\%}$). The experiments in the supplemental material show that the choice of database system can make a big difference in reducing this write-back bottleneck.

Figure 2.13(a), shows the same experiment for distributed chmod operations. The database layer can process the distributed chmod transactions with much lower latency since they do not require updating the primary key. Nonetheless, the writing of dirty data prior to transaction processing still dominates the latency. Figure 2.13(b) shows the throughput under varying mixes of multi-partition (MP) transactions, single-partition (SP) transactions and cache operations. Pure cache operations (100%) are 7x faster than SP transactions (100%). As soon as there are any MP transactions in a workload, even SP transactions that access the same data must be performed by the database layer. Therefore, there is more than a 1% drop in performance between 0%MP and 1%MP.

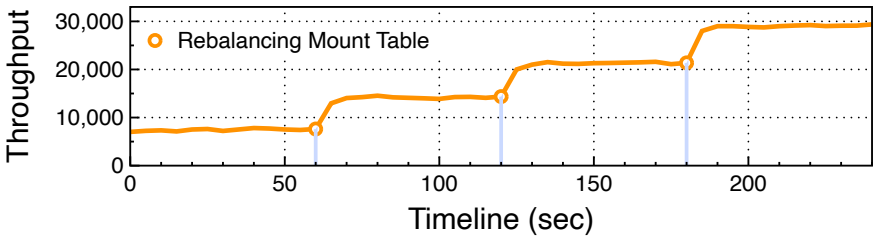


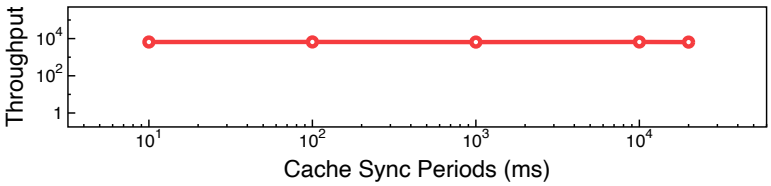
Figure 2.14: Hotspot Mitigation.

2.7.3.3 Hotspot mitigation

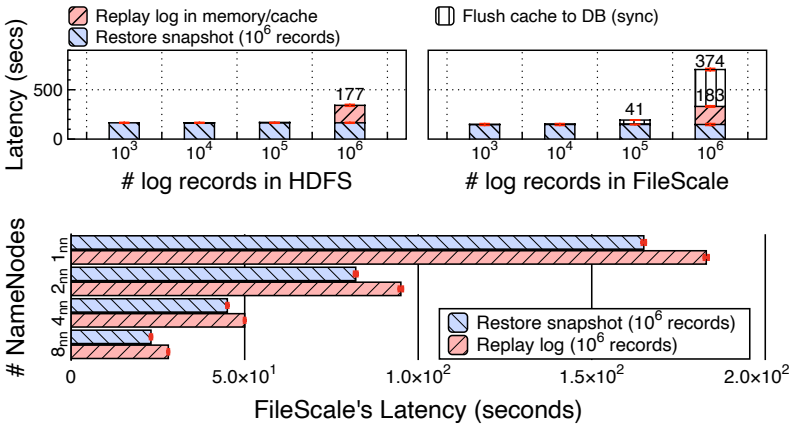
Figure 2.14 shows FileScale throughput as a hotspot is mitigated by rebalancing the mount table and distributing workloads across multiple NameNodes. We used benchmark utilities to create 4 subdirectories, all of which are initially mounted to NameNode 0. The proxy layer is triggered every 60 seconds to modify the mount table and assign each subdirectory to a new NameNode. The total throughput rises linearly as the hotspot workloads are re-distributed.

2.7.4 Disaster Recovery

Figure 2.15(a) shows that size of flush intervals do not impact system performance, since the writes to the database layer are asynchronous. In essence these overheads are pushed to recovery time. Therefore experiment with system restore to investigate this overhead. As described in Section 2.2, HDFS HA keeps its states (`FSImage` and `EditLog`) in a quorum-based storage so that a standby can take over quickly if the active NameNode fails. Similarly, the database snapshot of FileScale provides a transactional point-in-time consistent copy of all file metadata, and the separated logging system records every changes from the last snapshot.



Throughput when varying database sync periods.



Latency when recovering from its backups.

Figure 2.15: System restore operations.

Figure 2.15(b) compares the latency of restoring snapshots and replaying logs under the different system architectures. FileScale achieves comparable performance to HDFS when restoring 10^6 records from a snapshot. In HDFS, the NameNode only needs to replay edit logs in memory. However, FileScale must also update the database system after replaying logs in the cache. When this is done synchronously, this can add substantial latency to recovery. However, it can also be done asynchronously similar to how the database layer lags behind the cache layer in normal operations. In this experiment, as the number of NameNodes increases from 1 to 8 and file metadata spreads more evenly, FileScale's restore time decreases linearly.

2.7.5 The Impact of Database System Choice

FileScale uses a modular architecture such that any database system could be used for the database layer as long as it supports ACID transactions and provides an interface in which transactions can be submitted in SQL (with additional support for stored procedures). Most of FileScale's functionality is implemented using SQL at the database layer, which makes adding support for a new database system fairly straightforward. For example, the original version of FileScale was built over VoltDB's open source community edition, but when we were denied access to their enterprise version, we added support for Apache Ignite within a few weeks. In contrast, the rest of the FileScale codebase took two years of graduate student work.

Most metadata operations require asynchronous interaction with the database layer. Therefore, the choice of database system to use in the database layer often makes no runtime performance difference. However, when the cache layer does not have sufficient memory and cache misses are frequent, the performance of the database layer starts to matter. Furthermore, multi-partition transactions always require synchronous interactions with the database layer. We therefore explore the impact of different database systems under these scenarios in which the choice of database system becomes important.

2.7.5.1 Database Systems

VoltDB [53, 60] is an in-memory, high-throughput transactional database that eliminates many overheads of traditional disk-based systems, such as write-ahead logging, locking, latching, and buffer management [61]. It implements durability via a combination of asynchronous checkpointing and synchronous command logging in which all requests to the database are recorded in an input command log, and the system relies on its deterministic nature of transaction processing to ensure that replaying all input commands from a checkpoint results in an identical state as what it had been prior to the crash [62].

In VoltDB, the transactions can be implemented in pre-compiled, deterministic stored procedures. Each metadata operation in FileScale is thus designed within a VoltDB stored procedure that contains multiple SQL statements designed for that operation.

Apache Ignite [54] is an open-source distributed database for high-performance computing with in-memory speed. Ignite stores data in memory by default, but also includes an optional disk tier which we enabled for these experiments. Apache Ignite provides key-value APIs as well as MapReduce-like computations in addition to ANSI-99 SQL and ACID transactions.

2.7.5.2 Cache Misses

To understand the extent of the performance difference between using VoltDB vs. Ignite as the database layer for FileScale, we ran an experiment in which we in-

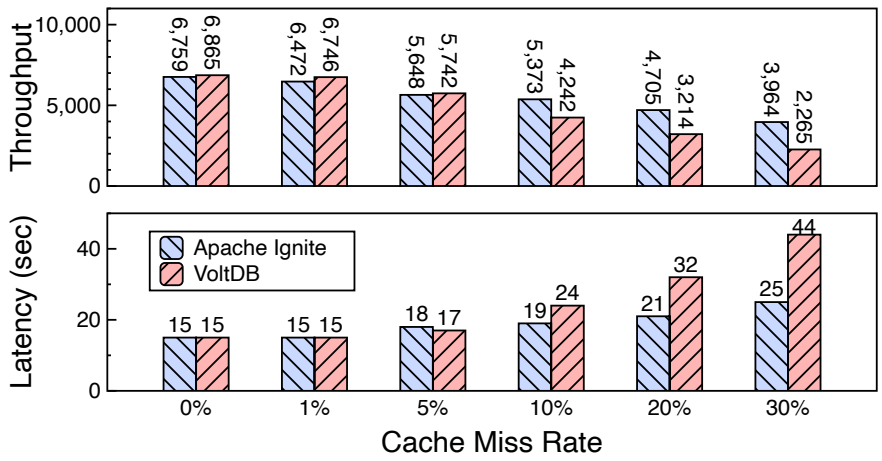


Figure 2.16: Cache miss penalty.

crease the cache miss rate of FileScale from 0% to 30% and measure the throughput reduction and latency increase on file create operations. For each experiment, we used 16 threads to operate on 100,000 files concurrently, and the results are shown in Figure 2.16. The throughput of FileScale with VoltDB and Ignite falls smoothly as the cache miss rate rises, and the latency rises gracefully.

VoltDB's performance declines faster than Ignite's after 10% cache misses. This is because Ignite's Key-value API `get()` can access the needed data from the storage using simple, light-weight access methods. In VoltDB, this was implemented using a standard SQL statement. Although VoltDB supports pre-compiling these SQL statements within a stored procedure, the performance of Ignite's lighter-weight key-value access methods is faster.

2.7.5.3 Multi-Partition Requests

As explained in Section 2.6.2, FileScale does a breadth-first search to find all dirty inodes of the subtree in the cached layer and bulk writes them to the

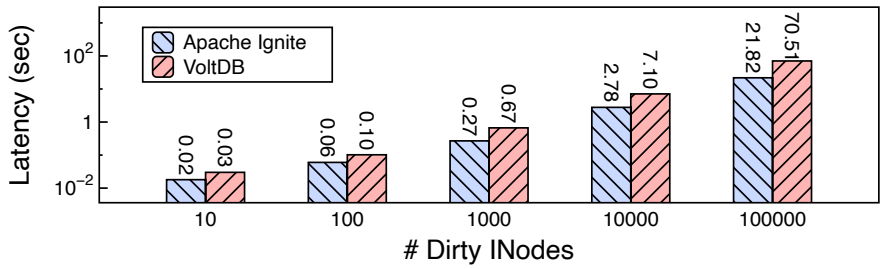


Figure 2.17: Dirty data flush penalty.

database. After the dirty inodes are written to the database, the entire multi-partition transaction is performed there. We saw in Section 2.7.3.2 that this write-back of dirty inodes is often the bottleneck for multi-partition transactions. Therefore, in this section, we investigate the performance of this cache flushing within the context of distributed transactions in more detail.

Cache Flushing: We first ran an experiment in which the amount of dirty inodes to be refreshed in database systems was increased from 10 to 100,000. Figure 2.17 shows the latency of writing dirty data to VoltDB and Ignite, with Ignite's bulk writes `putAll()` being approximately 3 times faster than the SQL statement `INSERT` in VoltDB as the number of dirty inodes increase.

Chmod: The pseudo code in Listings 2.1 and 2.2 depicts the differences in how the distributed chmod operation is implemented in the different database systems. We use the standard SQL APIs of VoltDB and Ignite to implement chmod operations. The first SQL query updates all of the children's permissions in a directory by matching the common ancestor path of the files.

The transaction is implemented using the `STARTS WITH <string-expression>` expression in VoltDB. Apache Ignite doesn't provide `STARTS WITH <string-expression>`

in its SQL API, so we use the syntactically equivalent `LIKE <string-expression>%` instead. The use of the `STARTS WITH` clause enables the utilization of available indexes, whereas `LIKE` requires a sequential scan, since the compiler cannot tell if the replacement text ends in a percent sign or not and must plan for any possible string value. This allows VoltDB to be slightly faster than Ignite in Figure 2.18 when no dirty data needs to be flushed. However, when the amount of dirty data that must be written back grows, Ignite outperforms VoltDB as we saw in Section 2.7.5.3.

```
# 1. Update all children in the subtree
UPDATE inodes SET permission = ?
  WHERE parent_name STARTS WITH ?;
# 2. Update the root inode of the subtree
UPDATE inodes SET permission = ?
  WHERE parent_name = ? AND inode_name = ?;
```

Listing 2.1: Distributed chmod operation in VoltDB.

```
IgniteCache<Object, Object> cache =
  ignite.cache("inodes").withKeepBinary();
// 1. Update all children in the subtree
cache.query(new SqlFieldsQuery("UPDATE inodes SET
  permission = ? WHERE parent_name LIKE ?%")
 .setArgs(permission, subtree_path)).getAll();

// 2. Update the root inode of the subtree
cache.query(new SqlFieldsQuery("UPDATE inodes SET
  permission = ? WHERE parent_name = ? AND inode_name = ?")
 .setArgs(permission, parent_name, inode_name)).getAll();
```

Listing 2.2: Distributed chmod operation in Apache Ignite.

Move: The pseudo code in Listings 2.3 and 2.4 shows the differences in how

the distributed move operation is implemented in VoltDB and Ignite[8]. The primary key of the inodes table is built up of parent and inode names. VoltDB and Ignite do not allow directly updating a primary key since the the data is partitioned by primary key (the partition is calculated using a hash function applied to the primary key's value), so changing the the primary key can cause the tuple to end up in the wrong partition. Thus, if a key needs to be updated it has to be removed and then re-inserted.

The move operation can be broken down into three parts: 1) Match the common ancestor path of inodes to obtain all fields of children inodes under the subtree. 2) For each child obtained from the first query, change the identifiers such as file path (parent name), inode name, and inode id. 3) Delete the old subtree from the database using the old primary key and commit the transaction.

```
# 1. Select children from the current subtree
SELECT inode_id, inode_name, parent_id, parent_name,
    permission, header, modification_time, ...
FROM  inodes WHERE parent_name STARTS WITH ?;

for (int i = 0; i < children.size(); ++i) {
# 2. Create new substree with new parent_name (in a loop)
INSERT INTO inodes(
    inode_id, inode_name, parent_id, parent_name,
    permission, header, modification_time, ...)
VALUES (?, ?, ?, ?, ?, ?, ?, ...);

# 3. Delete old subtree with old parent_name (in a loop)
DELETE FROM inodes WHERE
    parent_name = ? AND inode_name;
}
```

[8]For simplicity, unlike chmod operation, the corner case of the subtree's root inode is omitted here.

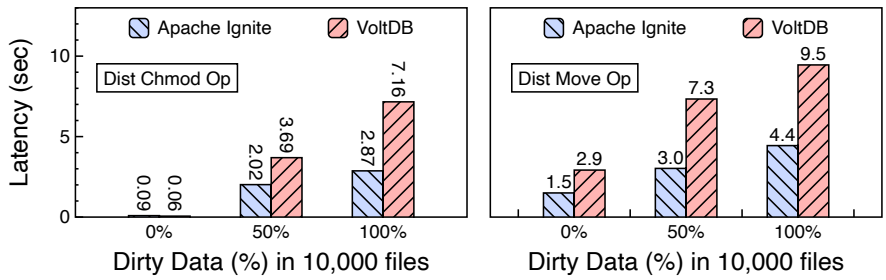


Figure 2.18: Distributed chmod and move operations.

Listing 2.3: Distributed move operation in VoltDB.

For steps 2 and 3, in VoltDB's stored procedure (Listing 2.3), we batch up multiple SQL statements in a loop by calling `voltQueueSQL()` for each statement then make a single call to `voltExecuteSQL()`. However, we eliminate the need for this slew of SQL statements in Ignite by leveraging its key-value APIs such as `putAll()` and `removeAll()` functions in Listing 2.4. Figure 2.18 (right) further shows that Ignite's distributed transaction for a move operation is around 2 times faster than VoltDB.

```
IgniteCache<Object, Object> cache =
  ignite.cache("inodes").withKeepBinary();

// 1. Select children from the current subtree
scan_query = new ScanQuery<>(
  new IgniteBiPredicate<Object, Object>() {
    @Override
    public boolean apply(Object key, Object obj) {
      return Key.field("parent_name").startsWith(old_parent);
    }
  }
);

List<Cache.Entry<Object, Object>> old_subtree
```

```
  = cache.query(scan_query).getAll();

// 2. Create new substree with new parent_name
Set<Object> keys = new HashSet<>();
Map<Object, Object> map = new HashMap<>();
ObjectBuilder KeyBuilder = ignite.binary().builder("key");
for (Cache.Entry<Object, Object> entry : old_subtree) {
  // new inode value
  Object new_inode = entry.getValue();
  new_inode = new_inode.toBuilder()
    .setField("parent_name", new_parent_name)
    .setField("parent_id", new_parent_id)
    .setField("inode_id", new_inode_id)
    .build();
  // new primary key
  Object new_key = KeyBuilder
    .setField("parent_name", new_parent_name)
    .setField("inode_name", inode_name)
    .build();
  keys.add(entry.getKey());
  map.put(new_key, new_inode);
}
cache.putAll(map);

// 3. Delete old subtree with old primary key
cache.removeAll(keys);
```

Listing 2.4: Distributed move operation in Apache Ignite.

2.8 Related Work

Merging file systems and database systems. There has been a large body of work which focuses on creating a hybrid system out of file system and database system components. For example, Murphy et al. demonstrated the potential of using a database system to store file system data and that the database-backed

file system is slower than a native file system by a small constant factor [63]. SQCK [64] employs declarative queries to check and repair a file system image , which improved functionality of `e2fsck`. BabuDB [30] uses log-structured merge (LSM) trees to replace the ext4-based backend at the metadata server. DeltaFS, TableFS, IndexFS and ShardFS [31, 32, 33, 34] store metadata in the local LevelDB instance [65] running atop a file system. BlueStore [35], a new backend for Ceph [66], stores metadata in RocksDB [67] as key-value pairs. However, all of this work focuses on improving the performance a single metadata server, rather than using the database system to implement scalable distributed transactions across partitions.

In industry, WinFS [68], (Windows Future Storage) brought the benefits of schema and relational databases to the Windows file system. Facebook Tectonic [69] delegates file system metadata storage to ZippyDB [70], a linearizable, fault-tolerant, sharded key-value store. However, ZippyDB only supports strong consistency for single-shard operations and does not support cross-shard transactions. Thus Tectonic only provides non-atomic cross-partition directory move operations. ADLS [45], (Microsoft Azure Data Lake Store) uses Paxos [71, 72] to maintain metadata that is stored in replicated Hekaton tables [73] and indexes. However, ADLS does not implement the cache layer of FileScale, and instead uses Hekaton directly by the NameNode, in which the quorum limits horizontal scalability and increases end-to-end latency. Colossus, the next-generation of GFS [20, 74], introduced a distributed metadata model using BigTable [48] which does not allow distributed transactions and thus does not allow multi-partition

metadata operations [46, 75].

High throughput distributed database systems were also used to scale file system metadata in the CalvinFS [43], HopsFS [44], and GiraffaFS [47] projects. CalvinFS uses a deterministic database system called Calvin [76] to store metadata, which supports high throughput distributed transactions. HopsFS uses a MySQL NDB cluster instead of Calvin, and shares FileScale's focus on being a drop-in replacement for HDFS. GiraffaFS uses HBase for a similar purpose. In contrast to these architectures, FileScale uses a three-tiered architecture to avoid synchronous interactions with the database system for requests that do not span multiple NameNodes (i.e. distributed transactions), which enables similar (or improved) performance relative to in-memory file system data structures. The performance consequences of these architectural differences were discussed in Section 2.7.

Federation. Giga+ [77] serves a similar purpose to FileScale's Proxy layer in that it divides each directory into a number of partitions that are distributed across multiple servers using hash functions. Giga+ uses a bitmap to map filenames to directory partitions and to a specific server. However, Giga must implement their own version of atomic multi-partition transactions and high availability, whereas FileScale gets these "for free" by leveraging the DB layer. This reduces the code maintenance obligations of the system, and enables FileScale to take advantage of advances in distributed transaction technology as the database community continues to make progress in this area. The View File System (ViewFs) [38] uses the client-side mount points to split HDFS into multiple physical namespaces

and presents a single virtual namespace to users. However, client configuration changes are required every time we add or replace a new mount point on ViewFs, and it is difficult to roll out these adjustments without affecting production workflows [78]. HDFS Router based Federation [39, 40] and ByteDance NameNode-Proxy [42] are extensions to ViewFS-based partitioned federation that uses routers forward client calls to the correct NameNode. But the router incurs network hops. The "watch mode" we called in Section 2.6 can update cached mount points automatically without sacrifice of network delay and consistency. Furthermore, none of these systems solve the general applicability limitation of HDFS Federation caused by partitioning the namespace.

Rebalancer [79] is a tool designed (under development) for RBF to move data to non-shared DataNodes among different subclusters. HDFS Federation Rename (HFR) [80] is a proposal in the Hadoop community that enables renaming files and folders across (federated) namespaces without migrating data on shared DataNodes through hardlinks. However, this approach requires the serialization of the entire directory subtree that is undergoing migration to external storage and then deserialization at the target NameNode.

DBOS. The DBOS system [81] proposes a data-centric operating system in which all operating system state is represented in a database system, and operations on this state are made via transactions initiated from stateless tasks. Many practical techniques that are implemented in FileScale can be applied to DBOS.

2.9 Summary

Although many file systems are capable of scaling to petabytes of data, scaling the metadata — specifically the number of unique files and directories — has been more of a challenge. This is because file systems expect extreme low latency interactions with metadata management, along with atomic, isolated, and durable guarantees and therefore typically store metadata in memory on a single machine. Recent attempts to scale metadata management via the use of distributed data stores that support distributed transactions have come at significant performance and efficiency costs, especially at low scale. In contrast, FileScale's architecture enables comparable performance to file systems that store all data in memory on a single machine at low scale, and yet can also scale linearly as the size of the metadata scales. Our experiments showed that FileScale can achieve multiple orders of magnitude superior performance relative to other approaches for scaling file system metadata. FileScale's architecture also enables elastic scaling of each layer in the architecture independently. For example, when cache misses start to become frequent, new nodes can be added to the caching layer to improve the performance of the deployment.

Chapter 3: Flock: A Practical Serverless Streaming Query Engine

3.1 Motivation

Many high-volume data sources, such as sensor measurements, machine logs, user interactions on a website or mobile application, and the Internet of Things, operate in real time. Stream processing systems are critical to providing the freshest possible data and driving organizations to make faster and better automated decisions. To provide widespread access to streaming computation, an ideal stream processing system must be performance competitive, scalable, highly available, easy to use and *low cost*.

Streaming jobs typically comprise multiple stages of execution organized as directed acyclic graphs (DAGs) based on their data dependencies, and each stage comprises several parallel tasks. These jobs show high variability and unpredictability, up to an order of magnitude more than the average load [5, 6]. This, along with the broad variety of user SLOs, makes statically configuring and tuning streaming systems extremely difficult. Furthermore, traditional server-centric deployments use clusters provisioned with a fixed pool of storage and compute resources to execute these jobs, it can frequently suffer from resource under- or over-provisioning, leading to resource wastage or performance degradation, re-

spectively.

The cloud benefits have driven many recent efforts to port streaming analytics applications to full managed streaming analytics services, e.g. Google DataFlow [82, 83, 84, 85] and AWS Kinesis Data Analytics for Flink [86]. These Backend as a Service (BaaS) serverless models are more elastic than on-premises alternatives and avoid their upfront costs. However, these cloud services provide elastic features that allow compute nodes to be added or removed dynamically, this scaling can take minutes, making it impractical on a per-query basis. In contrast, serverless platforms [7, 8, 9] fulfill the promise of transparent resource elasticity in the cloud [10, 11, 12]. Under the Function as a Service (FaaS) serverless model, developers and users decompose their applications into short-lived cloud functions. The ease of programming, fast elasticity, and fine-grained pricing in FaaS platforms allow for fine-grained scaling of resources to meet spiky demand, making them an appealing solution for streaming processing.

Compared to BaaS, the FaaS model provides more fine-grained elasticity with sub-second start-up times that can dynamically match the per-query basis with continuous scaling. Further, its billing methods are more fine-grained with millisecond granularity. For example, Kenesis service [86] is charged an hourly rate based on the number of Amazon Kinesis Processing Units (or KPUs) used to run the streaming application. However, on AWS Lambda [7], customers are only charged for the execution time they consume, often at a granularity of 1 ms [87]. Therefore a FaaS-based service is low cost to operate under low demand and can scale automatically to a high load at a proportional cost.

To explore the promise of function services for stream processing, we built Flock, a cloud-native streaming query engine that runs on FaaS platforms. Table 3.1 shows the differences between Flock and other state-of-the-art data analytics systems on FaaS. Existing approaches [13, 14, 15, 16] take advantage of the on-demand elasticity of cloud object storage services, such as Amazon S3 [17] to shuffle data, which increases the performance cost and compromises the advantages of a serverless system. Instead, Flock passes data through the invocation's payload between cloud functions. This is a general solution that can support Flock in multi-cloud platforms [7, 8, 9]. For example, current AWS Lambda limits are set at 6 MB for synchronous invocations, and 256 KB for asynchronous invocations [88]. The maximum HTTP request size for the 2nd iteration of Google Cloud Functions is 32MB [89]. The HTTP request length of Azure Functions is limited to 100 MB [90]. With payload invocations, Flock can store complete objects directly in query workflow state. This removes the need to read and write data from an external store service. Under the FaaS billing model, you do not pay for payload size but the cost of each job's duration, which is proportional to the aggregated runtimes across its component tasks – cloud functions. Therefore, this functional programming paradigm has a lower latency via storing data directly in the workflow, consequently, execution cost.

Payload invocation also eliminates the requirement for a query coordinator from the data architecture since Flock does not leverage any external storage service as a communication medium between functions, there is no need for a coordinator to monitor query stage completion and initiate new stages once depen-

dencies are met. Flock uses a unique way for passing many payloads/partitions to the same function instance, and shared data structures that ensure exactly-once aggregate of data on function services. When checkpointing is activated, query states are persisted upon checkpoints to guard against data loss and recover consistently.

We have implemented a prototype of Flock. We use this prototype to evaluate Flock by measuring its performance cost on the NEXMark [91] and Yahoo Streaming Benchmarks (YSB) [92] that include windowing functions. We find that under realistic deployment scenarios, compared with traditional streaming systems like Flink [93] deployed on EC2 instances, Flock is able to reduce costs more than an order of magnitude, with no observable effects on system throughput and query time.

Since the FaaS platforms manage allocation of compute resources across jobs, the goal of Flock is not to maximize resource utilization and enforce fairness but to reduce execution costs by increasing query performance and shortening function duration. Flock supports the vectorized processing on ARM processors, which brings 20% speedup and reduce costs by more than 30% on x86. Flock is thus the first streaming query engine, to the best of our knowledge, to support standardized abstractions, SQL and Dataframe API, on cloud function services, allowing users and engineers to avoid the time-consuming process of manually translating SQL into cloud workflows on heterogeneous hardware.

	SQL	SIMD	External Comm. Medium	Hardware	Client Coordinator	Type	Codebase
Locus [13]	No	No	ElastiCache, S3	x86_64	Yes	OLAP	Python
Lambada [14]	No	No	DynamoDB, SQS, S3	x86_64	Yes	OLAP	Python, C++
Starling [15]	No	No	S3	x86_64	Yes	OLAP	C++
Caerus [16]	Yes	No	Jiffy [94], S3	x86_64	Yes	OLAP	Python
Flock	**Yes**	**Yes**	**No**	**arm64**, x86_64	**No**	**Streaming**	Rust

Table 3.1: Comparison with Existing Serverless Data Analytics Systems.

3.2 Background

3.2.1 AWS Lambda

AWS Lambda [7] is a compute service that lets users run code without provisioning or managing servers. After uploading application code as a ZIP file or container image, Lambda automatically and precisely allocates compute execution power on a high-availability compute infrastructure and runs application code based on the incoming request or event, for any scale of traffic. When using Lambda, customers are responsible only for their code. Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources.

With AWS Lambda, users are charged based on the number of requests for their functions and the duration, the time it takes for application code to execute. Lambda counts a request each time it starts executing in response to an event notification or invoke call. Duration is calculated from the time user code begins executing until it returns or otherwise terminates, rounded up to the nearest 1 ms [87].

3.2.2 Apache Arrow and DataFusion

Apache Arrow [95] is a cross-language development platform for in-memory data. It specifies a standardized language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware with SIMD optimizations. Arrow was introduced in 2016 and

has since become the standard for columnar in-memory analytics, and as a high-performance interface between heterogeneous systems. Apache Arrow DataFusion [96] is an extensible query execution framework on the single machine, written in Rust, that uses Apache Arrow as its in-memory format. DataFusion supports both an SQL and a DataFrame API for building logical query plans as well as a query optimizer and execution engine capable of parallel execution against partitioned data sources.

The function executor in Flock is Arrow DataFusion, which has been extended to enable distributed query processing on cloud function services.

3.2.3 Streaming Query Processing

Any kind of data is produced as a stream of events, and it's most valuable at its time of arrival. Continuous queries are evaluated continuously as data streams continue to arrive, always reflecting the stream data seen so far. Since data streams are potentially unbounded in size, evaluating the query over different temporal windows of recent data from the streams is a common pattern. For example, in a tumbling window, events are grouped in a single window based on time or count. A event belongs to only one window. In a sliding window, events are grouped within a window that slides across the data stream according to a specified interval. Sliding windows can contain overlapping data; an event can belong to more than one sliding window.

Let's illustrate the semantics of queries in Flock by the following example.

Consider a hypothetical online auction system containing two tables:

```
CREATE TABLE Auction (id INT, item_name VARCHAR(128),
    description VARCHAR(255), initial_bid INT, reserve INT,
    date_time DATE, expires DATE, seller INT, category INT);

CREATE TABLE Bid (
    auction INT, bidder INT, price INT, date_time DATE);
```

The `Auction` table contains all items under auction, and the `Bid` table contains bids for items under auction. At some point the user executes a continuous query to determine the average winning bid price for all auctions in each category across a series of fixed-sized, non-overlapping, 10-second contiguous time periods[1]. In Flock, this query is expressed with the following DML:

```
-- Flock Context: Window::Tumbling(Schedule::Seconds(10));
SELECT category,
       Avg(final)
FROM   (SELECT Max(price) AS final,
               category
        FROM   auction AS A
               INNER JOIN bid AS B
                       ON A.id = B.auction
        WHERE B.date_time BETWEEN A.date_time AND A.expires
        GROUP BY A.id, A.category) AS Q
GROUP BY category;
```

When the user submits this query, it is continuously and transparently exe-

[1]We are assuming here that the auctions are very short-lived (with expiry time less than 10s) and that each auction starts and ends in a single window.

cuted in a microbatch mode on the cloud functions.

3.3 System Architecture

Flock is a cloud-native SQL query engine for event-driven analytics on cloud function services. Figure 3.1 illustrates the Flock's high-level architectural design. The cloud service provider packages and compiles the most recent query engine code into a single generic cloud function on a regular iteration cycle, then stores the resultant binary code in cloud object storage. When a user submits a SQL query, it is parsed, optimized, and planned as a series of low-level operators that the optimizer selects to execute the most efficient query. Flock breaks the execution plan into stages, with each stage consisting of a chain of operators with the same partitioning serialized as a string as part of the cloud function context. Flock creates cloud functions by using executable binary code from cloud storage and passing the encoded string (cloud context) as a function argument through the cloud vendor's SDK. The cloud function is created at the speed of light, and the query is processed in real time. Function arguments are deserialized as the cloud context during the initial instantiation of function instances, and therefore each function is customized for a specific set of parameters. The function is aware of carrying out a certain sub-plan and sending the result to the next function, allowing data flow in the cloud to occur without the intervention of a client coordinator.

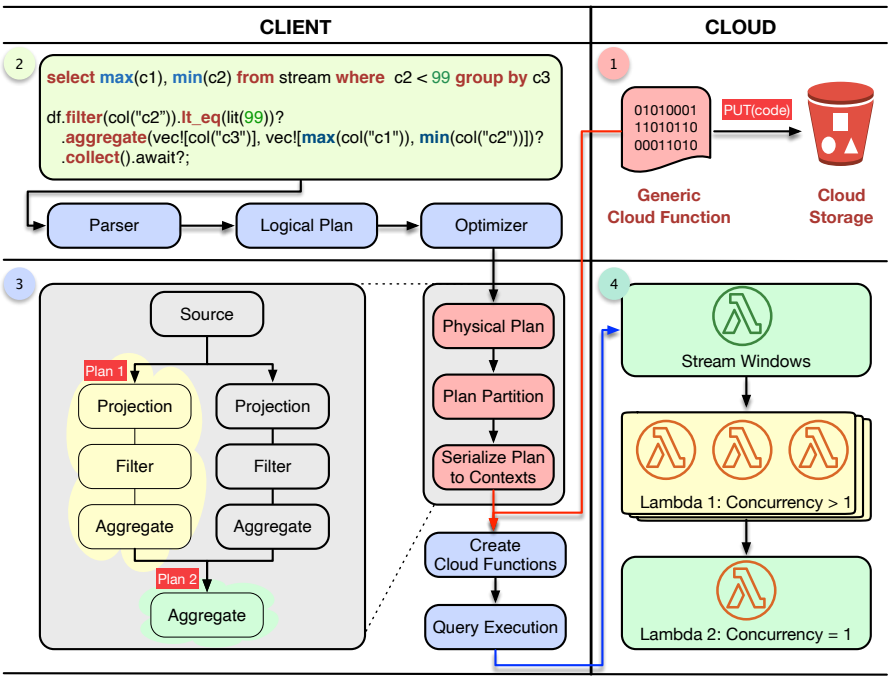


Figure 3.1: System Architecture.

3.3.1 SQL Interface

A query engine is a piece of code that can execute queries against data to produce answers to questions. Query engines provide a set of standard operations and transformations that the end-user can combine in different ways through a simple query language or application programming interface and are tuned for good performance. For example, SQL query engines are included in the most widely used relational databases, such as MySQL, Postgres, Oracle, and SQL Server. In addition, all data warehouses and lakehouses [1, 3, 97, 98, 99, 100, 101] come with a distributed SQL execution engine, such as Spark/Photon [102, 103], Flink [93], Presto/Trino [104], F1 [105], Impala [106] and Hive [107], for interactively querying massive datasets.

Some exploratory research has been done on doing data analytics on cloud

services [13, 14, 15]. However, there are yet no SQL-on-FaaS engines for data analytics. The end-user is compelled to split the physical plan for each query by hand when merging query stages into cloud functions as a dataflow execution paradigm on cloud. Forcing customers to use cloud vendor lock-in APIs to orchestrate query stages has the same effect as forcing users to create query execution plans directly in database systems. The user plans may be suboptimal, result in significant performance loss, and such customized directionally-acyclic graphs (DAG) are error-prone and are rarely to be reused. Furthermore, some cloud customers have raised concerns about vendor lock-in, fearing reduced bargaining power when negotiating prices with cloud providers. The resulting switching costs benefit the largest and most established cloud providers and incentivize them to promote complex proprietary APIs that are resistant to de facto standardization. Standardized and straightforward abstractions, such as SQL and Dataframe API supported by Flock, would remove serverless adoption's most prominent remaining economic hurdle.

3.3.2 Distributed Planner

Flock uses rule-based optimizations to apply predicates and projection pushdown rules to a query plan that executed against the logical plan before the physical plan is created (see Figure 3.1). The physical plan is broken into a DAG of query stages in the client-side, where each stage consists of a chain of operators with the same partitioning. Each query stage is assigned to a cloud function using

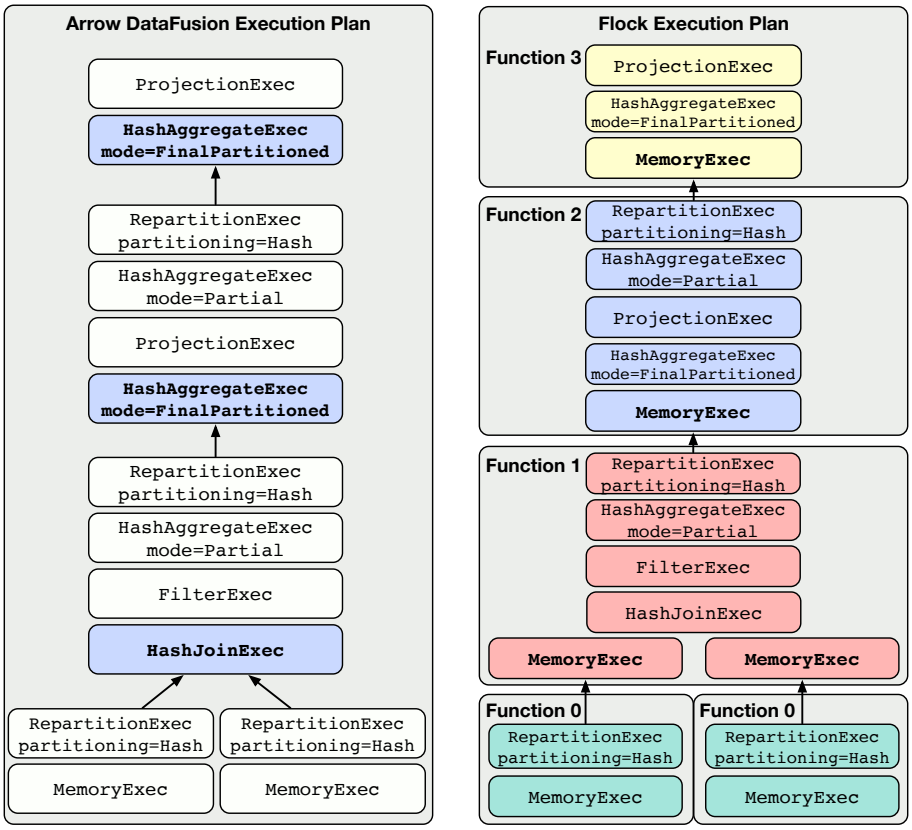


Figure 3.2: Physical Plan Partition.

template specialization approach described in the next section. A directed edge from one stage to another represents data flow between cloud functions.

Figure 3.2 shows the plan partition of the query example in Section 3.2.3. The physical plan of Arrow DataFusion is a nested layout in memory in which the data sink, not the data source, is the root reference. Flock uses top-down breadth-first algorithm to split the physical plan into a DAG of query stages. Flock separates the plan when it encounters aggregate[2], join and sort operations so that data can be shuffled around between query stages. Each stage or subplan deletes the old reference to the child plan and replaces it with an empty `MemoryExec` that

[2]"`HashAggregateExec`: mode=partial" is the same as doing partial aggregate within individual cloud functions or workers, and there is no requirement for plan partition.

has the same schema as the child plan. `MemoryExec` represents the execution plan for reading in-memory batches of data. When the current stage receives all the output of the previous stage, it will feed the data to its `MemoryExec` and complete the query execution. Furthermore, while splitting the physical plan, Flock creates a corresponding cloud context for each query stage in order to make the dataflow paradigm operate on cloud function services.

3.3.3 Microbatch Execution Mode

Flock runs in a micro-batch execution mode, similar to Apache Spark's Structured Streaming [102, 108, 109], that processes data streams as a series of micro batch tasks, achieving exactly-once fault-tolerance guarantees. In this mode, epochs are typically set to be a few hundred milliseconds to a few seconds, and each epoch executes as a traditional analytical job composed of a DAG of functions. When compared to continuous operator model [110, 111, 112], micro-batch and FaaS are more natural fits. There are two main reasons for this: 1) The cloud function is billed based on the number of invocations and duration, whereas record-by-record is many orders of magnitude more expensive. 2) Some cloud providers, e.g. AWS Lambda, only allow the function instance to execute one request at a time, and a huge number of requests (via record-by-record) causes the function's latency to rise dramatically.

During query planning, Flock automatically chains together sequences of functions, each of that corresponds to a query stage. Flock implicitly invokes the

first cloud function to trigger the execution workflow at recurring times. Although all created functions have exactly the same binary code, when a function is instantiated in the cloud, its environment variable contains the specific cloud context carried when it was created. Therefore, different function instances can be specialised through the context (see Section 3.4). Functions share states by passing arguments/payloads and return values to each other, which does not incur any additional costs. The main challenge is determining how to send the shuffled states to the same function instance without the need of an external communication medium. We accomplished this by setting the stateful function's concurrency to **one** and allocating global memory that allows the function to reuse "static context" across multiple invocations to the same instance. More details on how to mitigate hotspots are described in Section 3.5.5.

3.3.4 Fault Tolerance

3.3.4.1 State Management

Flock achieves fault tolerance through the employment of a write-ahead log and a state store. Both of them run over object storage system such as S3[3] to allow parallel access. 1) the log keeps track of which data has been processed from each input source and reliably written to the output sink. 2) the state store holds snapshots of operator states for aggregate functions. Simiar to Spark Streaming [109], states are written asynchronously, and can be behind the latest data written to the

[3]Starting from Dec 2020, all S3 `GET`, `PUT`, and `LIST` operations are now strongly consistent [113].

output sink. In the event of a failure, the system will automatically track whatever state it last updated in its log and recompute state from that point in the data.

Input sources like Kafka [114] and Kinesis [86] are replayable that allow re-reading recent data using a stream offset. The function writes the start and end offsets of each epoch durably to the log. The stateful functions/operators regularly and asynchronously write the epoch ID along with their state checkpoint to the state store, utilizing incremental checkpoints if possible. These checkpoints aren't required to occur every epoch or to block processing. Upon recovery, the new function instance starts by reading the log to find the last epoch that hasn't been committed to the sink, including its start and end offsets. It then uses the offsets of earlier epochs to reconstruct the states in memory from the last epoch written to the state store. Finally, the system reruns the last epoch, and then executes the micro-batch from the new epoch.

3.3.4.2 Invocation Failure

If a cloud function times out or is terminated, the computed end-result is accurate, with no data loss. This is because the new function is resumed using the most recently stored checkpoint and states from S3. However, unlike traditional nodes, function invocation errors can occur when the invocation request is rejected by issues with request parameters and resource limits or when the function's code or runtime returns an error. If the asynchronous invocation fails, Lambda retries the function since the payload is part of the invocation and hence no data

is lost. When an event fails all processing attempts or expires without being processed, it's put into a dead-letter queue (DLQ) [115] for further processing, which is part of a function's version-specific configuration. If the synchronous invocation fails, Flock implements a linear backoff algorithm for automatic retries (see Section 3.5.3).

The function may receive the same request/payload multiple times for asynchronous invocation because Lambda's internal queue is eventually consistent [116]. The stateful function maintains a bitmap to avoid double-counting and to ensure each payload is aggregated and processed exactly once (see Section 3.6.0.2).

3.4 Function Templates

3.4.1 Template Specialization

Legal cloud functions are only scripts or compiled programs, so many systems [15] embedded the physical plan to the function code during the code generation phase. They generate code for the individual tasks, compiles it and packages it with necessary dependencies. To execute a job, a scheduler launches tasks as serverless functions and monitors their progress. However, compiling cloud functions and dependencies at query runtime might cause delays of seconds or even minutes, slowing query response time. For example, Flock is a Rust-based cloud-native query engine; if we build an `x86_64-unknown-linux-gnu` release version with the features SIMD and mimalloc/snmalloc [117, 118] on an AWS EC2 instance – c5a.4xlarge, the build time was roughly 4 minutes even with incremental

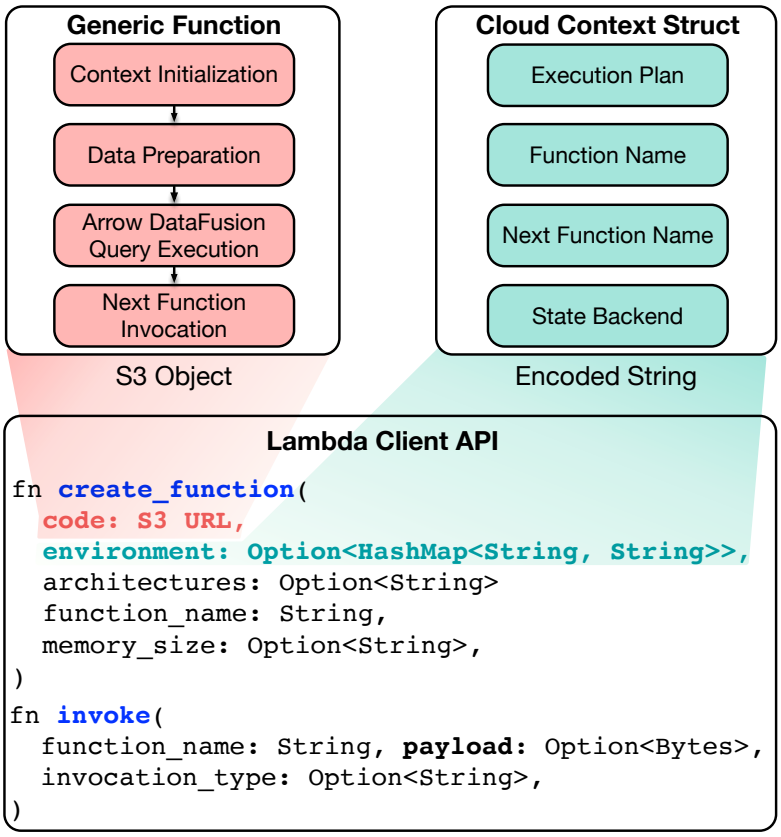


Figure 3.3: Generic Function and Template Specialization.

compilation. What's worse, it takes 8m 33s to build from scratch. This is because Flock requires a lengthy dependency tree to be built [119]. Another approach is that Locus [13] is built on Pywren [120, 121], a pure Python implementation that omits the code-generation and compilation steps and directly takes task code and execution plan as input with sacrificing performance and cost (longer charged duration).

We propose function template specialization as a way to completely *eliminate* the compilation stage from the query execution pipeline. Template specialization in programming languages allows alternative implementations to be presented based on specific properties of the parameterized type that is being instantiated,

enabling for certain types of optimization and reducing code bloat. Similarly, as shown in Figure 3.3, Flock's service provider creates, builds, and archives a **generic** cloud function as a `bootstrap.zip` file, which is primarily made up of four components: cloud context initialization, data collection and preparation, query execution and next function invocation (more details will be explored in Section 3.4.2). The service provider then uploads the `bootstrap.zip` to the cloud object storage, such as Amazon S3 [17], and makes new public release available for Flock users.

Flock eliminates the requirement for the client or central registry to spend time compiling SQL execution plans and new cloud functions into binary code, resulting in much lower end-to-end latency[4]; Flock creates functions right away using the S3 object of the generic function that the service provider has provided in advance [124]. In addition, the cloud context, which includes the execution plan, is serialized and sent as a string into the Lambda API `create_function()`'s parameter `environment` in Figure 3.3. The cloud context is compressed with Zstd [125] after serialization by default since Lambda environment variables have a default 4 KB service quota that cannot be raised [126]. In that case, Flock can store the execution plan in S3 and preserve the S3 object key in the cloud context if the execution plan exceeds this limit. The environment variable settings that are accessible from function code during execution on cloud. The query launch time is reduced by 10,000 times using this approach since launching a cloud function just

[4]For developers who want to write custom stream processing logic, Flock's stateful operators are UDFs with state that still require users to compile function code during query runtime. In this case, JIT code generation for each query over LLVM [122] or Cranelift [123] is a better solution to reduce branching overhead and the memory footprint.

requires the creation of a function without compilation.

Flock then invokes the newly created function name to execute the query on the cloud function services via the Lambda API `invoke()` [127]. The context initialization is performed once per function instance to prepare the cloud environment for invocations; it reads the encoded string from the environment variable and deserializes it as the cloud context. The generic function template specialization is then achieved. Even while all functions have the identical code — generic template, each function can identify the specific execution plan and which function to deliver the output to via the cloud context when it is instantiated in the cloud.

3.4.2 Generic Function

Flock is a new generation of cloud native query engine that consists simply of generic functions and a client library. The generic function can work with any type, rather than a specific type only, allowing it to be designed, built, and delivered to the cloud platform ahead of time. To take use of the latest query engine capabilities, the cloud service provider only has to offer customers with an updated version of the generic function on a regular basis without disclosing the source code. The client library can translate SQL queries to executable cloud functions.

A generic function is a function code whose behavior depends on the identities of the arguments supplied to it via `environment` (see Figure 3.3). When a

function is invoked, it deserializes the cloud context given by the client to discover the appropriate code regions — those with specializers that are compatible with the actual context.

The pseudo code in Listings 3.1 shows how the generic cloud function is implemented and operated. The function code can be broken down into four parts:

(1) *Cloud Context Initialization.* `INIT` (line 5) is a synchronization primitive for running a one-time global initialization. The given closure `ctx_fn` (line 9-14) is used to deserialize environment variables into cloud context, and it will be run if `call_once` (line 15) is used for the first time; otherwise, the routine will not be invoked. Private data that is only used per invocation should be defined within the handler. Global variables such as `CLOUD_CONTEXT` retain their value between invocations in the same execution environment. As a result, the cloud context is only initialized once throughout the lifetime of the instance, and future invocations reuse the resolved static context. `arena` (line 11 and 23) is a type of global resource that are created during initialization stays in memory between invocations, allowing the handler to collect states across invocations. We explain it in more details in Section 3.6.0.2.

(2) *Data Preparation.* The function essentially receives the payload in JSON format from the HTTP request's body, computes the result, and either returns it to the client or forwards it to the next functions as HTTP requests. When the runtime receives an event, it passes the event (line 22) to the function handler. Flock leverages Apache Arrow [95] to save streaming data (line 24) in the in-memory columnar format to maximize cache locality, pipelining and SIMD instructions on

modern CPUs. In the case of the function associated with the aggregate operation, such as `HashAggregateExec`, Flock uses `Arena` to collect all data partitions before being given to the embedded query engine in the current function. More details are described in Section 3.6.0.2.

(3) *Query Execution.* The function includes Arrow DataFusion [96], an in-memory query engine that provides both a DataFrame and SQL API for querying CSV, Parquet, and in-memory data. DataFusion leverages the Arrow compute kernels for vectorized query processing. All rows with a particular grouping key are in the same partitions, such as the case with hash repartitioning on the group keys. Data partitions are processed in parallel in the cloud function (line 26).

(4) *Next Function Invocations.* Following the execution of the query stage in the current function, the output is placed into the next function invocation's payload (see `invoke()` in Figure 3.3), and finally, a synchronous or asynchronous invocation (line 27) is made to make distributed dataflow possible. The implicit invocation chain is analogous to functional programming. More complex data shuffling are described in detail in Section 3.6.0.3.

```
1 use lambda_runtime::{service_fn, LambdaEvent};
2 use serde_json::Value;
3
4 /// Initialize the function instance once and only once.
5 static INIT: Once = Once::new();
6 static mut CLOUD_CONTEXT = CloudContext::Uninitialized;
7
8 macro_rules! init_cloud_context {
```

```
    let ctx_fn = || match std::env::var(&**CONTEXT_NAME) {
        Ok(s) => { CLOUD_CONTEXT = CloudContext::Lambda((
            ExecutionContext::unmarshal(&s), Arena::new()));
        }
        ...
    };
    INIT.call_once(ctx_fn);
    match &mut CLOUD_CONTEXT {
        CloudContext::Lambda((ctx, arena)) => (ctx, arena),
        CloudContext::Uninitialized => panic!("uninitialized!"),
    }
}

async fn handler(event: LambdaEvent<Payload>) -> Result<Value> {
    let (mut ctx, mut arena) = init_cloud_context!();
    let (input, status) = prepare_data(ctx, arena, event)?;
    if status == HashAggregateStatus::Ready {
        let output = collect(ctx, input).await?;
        invoke_next_functions(ctx, output, ...).await
    } else if status == HashAggregateStatus::NotReady {
        Ok(json!("response": "data is not yet ready"))
    } else if status == HashAggregateStatus::Processed {
        Ok(json!("response": "data has been processed"))
    }
}

#[tokio::main]
```

```
async fn main() -> Result<()> {
    lambda_runtime::run(handler_fn(handler)).await?;
    Ok(())
}
```

Listing 3.1: Generic Function Skeleton.

3.4.2.1 Heterogeneous Hardware

According to the AWS blog [128], AWS Lambda functions running on Graviton2 [129], using an Arm-based processor architecture designed by AWS, deliver up to 34% better price performance compared to functions running on x86 processors for a range of serverless applications including real-time data analytics. To give users with better price-performance, Flock provides function binaries for both x86 and Arm architectures, and users may select different generic function binaries from AWS S3 bucket to create Lambda functions that operate on x86 and/or Arm processors. Currently, Flock has 4 versions on S3 — `x86_64-gnu`, `x86_64-musl`, `aarch64-gnu` and `aarch64-musl`.

For Lambda functions using the Arm/Graviton2 processors, duration charges are 20% lower than the current pricing for x86. However, the reported performance difference (19%) between x86 and Arm by AWS may not include SIMD optimization. An unanswered question is who performs better on query operations when AVX2 and Arm Neon intrinsic are employed. The Graviton2 processor also has support for the Armv8.2 instruction set. Armv8.2 specification includes the large-system extensions (LSE) introduced in Armv8.1. LSE provides low-cost

atomic operations and improves system throughput for CPU-to-CPU communication, locks, and mutexes. To measure the difference between architectures, we compared the latency and duration cost between the two architectures in Section 3.7.

3.5 Serverless Actors and Communication

The actor model is a highly popular computational pattern, which simplifies the job of composing parallel and distributed executions by using a basic unit of computation: the actor. Flock is an actor model that provides an isolated, independent unit of compute and state with multiple-threaded execution on cloud functions for serverless event-stream processing service with pay-for-use.

3.5.1 One-way Communication

The most important element of the actor model is that actors can communicate via asynchronous messages. Previous work has proposed solutions for data exchange in the serverless context [13, 14, 15, 130, 131]. They rely on external storage to exchange large amounts of data since cloud functions can't accept incoming connections. For example, Starling [15] uses Amazon S3 to pass intermediate data between function invocations. However, the solutions consist of additional services, which increases the latency (I/O), billed expenses of function duration and S3 access, and therefore compromises the advantages of a serverless system.

Flock differs from earlier systems in that it is built for real-time stream pro-

cessing on gigabytes of data rather than OLAP workloads. AWS Lambda functions have a 6 MB payload size limit for synchronous invocations and a 256 KB size limit for asynchronous invocations [88]. The function's concurrency is the number of instances that serve requests at a given time, and the default regional concurrency limit starts at 1000 [132] which can be easily increased to 5000 by contacting Amazon. By combining these two AWS Lambda quotas above, as well as data encoding and compression, Flock can transfer GB-level intermediate results between functions without using external storage.

When a function is invoked (see `invoke()` in Figure 3.3), Flock passes data in the payload and the payload is serialized to JSON bytes because the content type of HTTP requests body is enforced to `application/json` in AWS Lambda. Data partitioning guarantees that each partition can be placed in a function's payload, and the functional chain seamlessly passes the data to the next query stage. This removes the need to persist and load data from data stores such as DynamoDB [133] and S3 [17]. There is no additional cost associated with invoking Lambda functions with a payload, which reduces bill cost and duration.

Table 3.2 shows the latency difference between AWS S3 and function payload. By default, objects are compressed with Zstd [125], which provides a 4x compression ratio on NYC Citi Bike trip data [134]. Therefore, the real single partition size we tested reached up to 60MB, which is enough to handle streaming workloads. The latency in the table also includes the overheads of marshalling/unmarshalling and compression/decompression, this is because serialization and deserialization phases happen in Lambda Rust Runtime [135]. However, these

Object (Zstd)	S3 Read	S3 Write	S3 Total	Lambda Sync	Lambda Async
1.5 KB	0.471	0.113	0.584	0.020	0.030
15 KB	0.471	0.144	0.615	0.020	0.044
150 KB	0.653	0.205	0.858	0.036	0.066
1.5 MB	1.615	0.594	2.209	0.281	0.785
15 MB	11.720	1.828	13.548	2.201	6.054

Table 3.2: The latency comparison (seconds).

parts are not bottlenecks, accounting for less than 7% of total time. When the compressed partition is less than or equal to 1.5MB, the payload communication is an order of magnitude faster than S3. Since 15 MB exceeds the maximum size of the function payload, Flock executed the same function instance three times synchronously or 60 times asynchronously, resulting in a 6x or 2x improvement, respectively. We'll explain how multiple payloads are routed to the same running instance in Section 3.5.5, which is the critical part of data shuffling.

The limitation of this approach is that AWS Lambda does not yet provide per-instance concurrency like GCP Functions [136] for now. GCP function allows for up to 1,000 concurrent requests on a single instance of an application, providing a far greater level of efficiency. In extreme cases, multiple data partitions shuffling for aggregation may performs poorly when constrained to a single-request model.

3.5.2 Sync and Async

When a function is invoked asynchronously, Lambda puts the event in a Lambda-owned queue and returns right away, rather than exposing Lambda's

internal queues directly. A separate process reads events from the queue and executes the function. AWS Lambda is a multitenant system that implements fairness is by setting per-customer rate-based limits, with some flexibility for bursting [137]. Therefore, there may be an occasional invocation delay while dealing with heavy workloads.

Figure 3.4 shows the benefit of asynchronous calling that is when the current function invokes the next function, it can return immediately instead of waiting for the succeeding function to complete its execution. This significantly decreases the expense of duration. If n represents the total number of query stages, f_i represents the lambda function or function group corresponding to the ith query stage. The total asynchronous invocation cost of the bill is

$$\sum_{i=0}^{n} \lambda(f_i) + \sum_{i=0}^{n} d(f_i)$$

$\lambda(f_i)$ indicates the cost of function invocations to the ith query stage with a specific memory and processors. $d(f_i)$ is the billed duration cost of the ith query stage. For the synchronous invocation, the duration cost (including waiting time) of the ith stage is $\sum_{j=i}^{n} d(f_j)$. The total cost of the bill is

$$\sum_{i=0}^{n} \lambda(f_i) + \sum_{i=0}^{n} \sum_{j=i}^{n} d(f_j)$$

However, compared to asynchronous invocation, synchronous invocation is faster and more reliable, and it won't be affected by internal queue throttling. Furthermore, if the query is executed by a single function or the stages are shal-

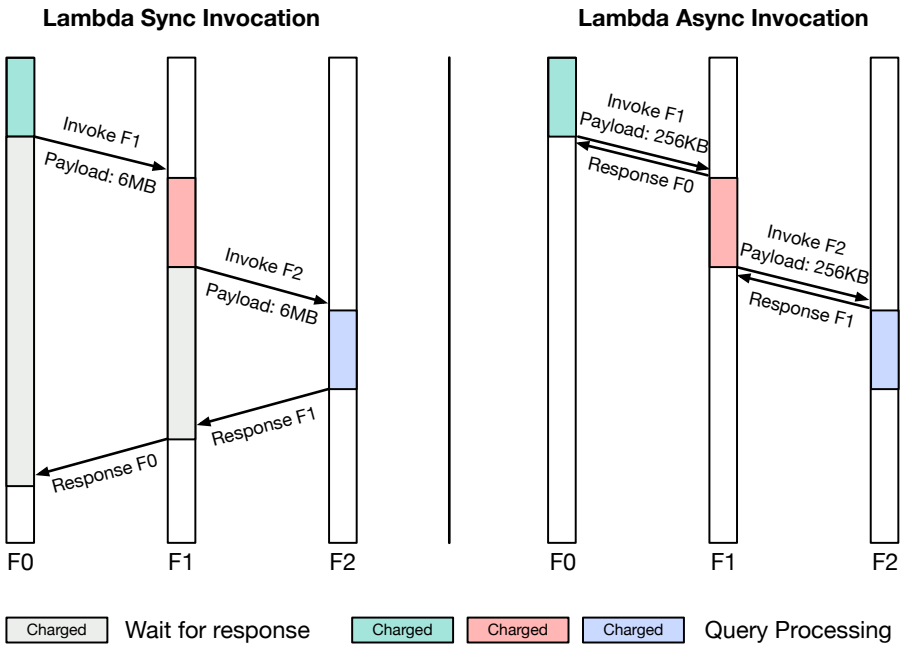


Figure 3.4: The duration charge comparison.

low, the billed duration of synchronous invocation is lower and less expensive. This is because its payload maximum is 6MB, which is 24 times larger than the asynchronous's 256KB, the synchronous approach requires 24 times fewer invocations, alleviating the the single-request model problem of AWS Lambda. Since each function call is charged for duration, the asynchronous approach may have a higher cost in terms of invocation and duration time.

3.5.3 No Coordinator

Migrating streaming applications from a traditional serverful deployment to a serverless platform presents unique opportunities. Traditional serverful deployments rely on existing workflow management frameworks such as MapReduce [138], Apache Spark [102, 109], Sparrow [139], Apache Flink [93] to provide

a logically centralized scheduler for managing task assignments and resource allocation. The scheduler traditionally has various objectives, including load balancing, maximizing cluster utilization, ensuring task fairness, keeping track of distributed tasks, deciding when to schedule the next task (or set of tasks), and reacting to finished tasks or execution failures. A traditional serverful scheduler is not required by serverless computing. This is because FaaS providers are responsible for managing the containers or MicroVMs [140] and serverless platforms typically provide a nearly unbounded amount of ephemeral resources. However, existing data systems on FaaS platforms like Starling [15] and Lambada [14] **still** require a coordinator to monitor task completion and start new stages once dependencies are completed due to they use S3 as the communication medium between functions. They would otherwise have no way of knowing if the current query step is complete.

Flock *completely eliminates* the coordinator by putting the function name of the next stage in the current function's cloud context during the query planning phase on the client-side (see Figure 3.3). When the current function finishes the computation, it simply passes the result to the next function invocation's payload. 1) For asynchronous invocation, if the function is terminated abnormally or thrown invocation errors, AWS Lambda retries the function. Flock configures a dead-letter queue [115] on the function to capture events that weren't successfully processed for further processing. 2) For synchronous invocation, Flock implements truncated linear backoff algorithm that uses progressively longer waits between retries for rate limit exceeded errors. The retries are only required in

synchronous invocations when Flock passes multiple payloads to a single function with concurrency equals to 1, more details are explained in Section 3.5.5. The current function, in this approach, regularly re-invokes a failed function, increasing the waiting time between retries until the maximum backoff time is reached. The following is the duration of the wait:

$$min(50 * \text{increase\_factor} + \text{random\_milliseconds}, \text{max\_backoff})$$

The increase_factor starts from 1, and reset to 1 when 50*increase_factor exceeds the maximum backoff. random_milliseconds is bounded to 100ms that helps to avoid cases in which many functions retry at once, sending requests in synchronized waves. In conclusion, removing the query engine's core coordinator makes coding, operation, and maintenance easier while potentially reducing query processing time.

3.5.4 Function Name

The name of the cloud function is made up of three parts:

```
Function Name: <Query Code>-<Query Stage ID>-<Group Member ID>
```

The query code is the hash digest of a query. The query stage id is a 2-digit number that represents the position of a stage in the DAG, and the group member id is the position of the function in a group it belongs to. The function name does not include a timestamp so that the created function can be reused by continuous queries without incurring a cold start penalty. This naming convention guarantees

that each cloud function is appropriately identified and categorized into a distinct query, allowing Flock to detect and resolve issues efficiently.

The cloud function concurrency is the number of instances or execution environments that serve requests at a given time [141]. The first time you invoke a function, AWS Lambda creates a function instance and runs its method to process the event. After the invocation has ended, the execution environment is retained for a period of time. If another request arrives, the environment is reused to handle the subsequent request. However, if requests arrive simultaneously, Lambda scales up the function instances to provide multiple execution environments, and the events are processed concurrently. Each instance has to be set up independently, so each instance experiences a full cold start. Flock uses Lambda SDK API [142] to set the maximum concurrency for each function in the query DAG, ensuring that the function has the ability to scale on its own, preventing it from growing beyond that point.

3.5.5 Function Group

Flock sets the default 1000 concurrency to stateless (non-aggregate, e.g. scan, filter and projection) functions. Each of them is preferentially executed on a data partition contained the same keys to maximize data parallelism. If the concurrency of the stateful (aggregate, e.g. group by, sort and join) function is set to more than 1, Flock is unable to ensure the integrity of the query results. Lambda is likely to spawn multiple running instances to handle payloads from the non-

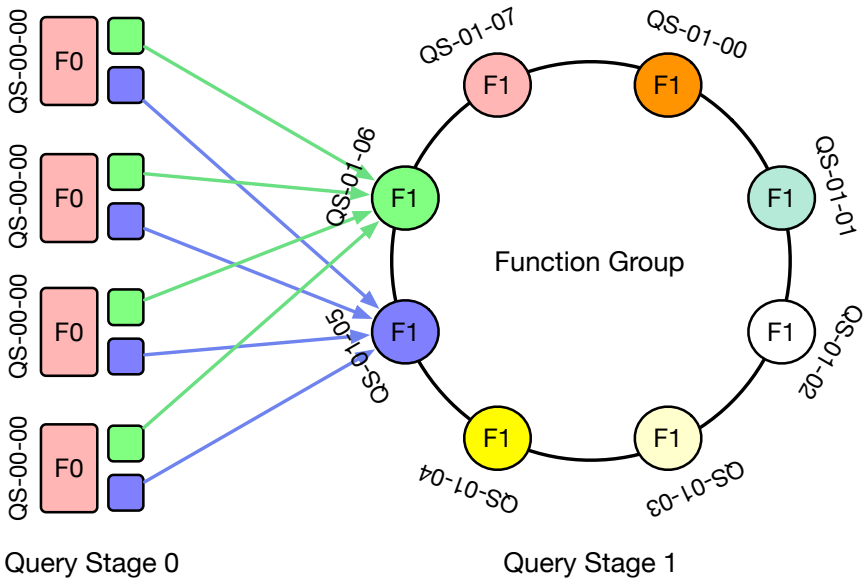


Figure 3.5: Cloud Function Group.

aggregate functions, causing the partial results to divergence and ultimately fail to aggregate. Therefore, for aggregate function, Flock set its concurrency to 1, which enforces AWS Lambda to create at most one running instance for the aggregation function at any given time.

However, one of the key benefits of serverless query engine is the ease in which it can scale to meet traffic demands or requests, with little to no need for capacity planning. Setting the concurrency of aggregate functions to one goes against the essence of serverless computing. Because there is only one function instance for the current query stage, and AWS Lambda does not yet provide per-instance concurrency [136] like GCP Functions to accept concurrent requests on a single running instance, hot spots are caused by awaiting the completion of the preceding aggregate task.

We proposes the *function group* technique that creates a set of cloud functions

in a group for each query stage after physical plan partition to reduce the hotspot effect. 1) for non-aggregate function, its concurrency is 1000 by default. Flock only makes one function member (name) in that group, and AWS Lambda governs its running instances and routes requests to them. 2) for aggregate function, its concurrency is set to 1. Flock creates a group consisting of multiple identical functions with different names.

Figure 3.5 shows an example of how the cloud function group works. The function 0 contains the query stage 0 and has the default 1000 concurrency. Lambda starts four instances of function 0, each of which conducts a local aggregation and hash partitioning of the output into two distinct payloads – green and blue boxes. The same color payload in different function instances includes the same shuffle id (see Section 3.6.0.1) that is used to generate a deterministic random key to perform consistent hashing [143]. Because the cloud context of the function 0 (see Figure 3.3) has the next function name — here is `Group(QS-01,8)`, `QS-01` is group name and 8 is group size. Using the group information, Flock maps payloads associated with the same key on four distinct instances to the same function name (same instance) in the next stage. This approach distributes shuffling operations to multiple function instances and guarantees data integrity since all functions' concurrency in the group equals to 1. To minimize serial aggregates on the same function instance caused by hash collisions, each function 0 does a single consistent hash lookup to determine the ring's start point, then maps different data partitions to different function names in a counterclockwise order. For example, if green partitions are mapped to `QS-01-06`, then blue partitions is mapped to

QS-01-05.

With consistent hashing, on the other hand, the hashing function is independent of the number of cloud functions in the group. This allows the extended optimizer to dynamically routing the data as we add or remove functions, and hence, scale on demand to balance hotspots and cold starts[5]. Furthermore, by reading statistics from the state store, function instances can agree on dynamically coalescing shuffle partitions for adaptive query execution [144].

3.6 Flock Dataflow Paradigm

For workloads using streaming data, data arrives continuously, often from different sources, and is processed incrementally. The processing function does not know when the data stream starts or ends. Consequently, this type of data is commonly processed in temporal windows. Flock has native support for tumbling, sliding and session window functions, enabling users to launch complex stream processing jobs with minimal effort. The first query stage is datasource functions, which continue to fetch messages from the stream until a full batch is obtained or the time window expires.

Flock's execution plan, unlike traditional distributed execution engines, is a dynamic directed acyclic graph that changes with time in the cloud rather than a static one on-premise. This is because each stage in the query DAG represents a cloud function group, AWS Lambda automatically scales up and down running instances for each stage based on the number of incoming events. However, as the

[5]We have yet to implement this feature in the Flock codebase.

data shuffling is sent in pieces via the function invocation's payload, the aggregate function is likely to receive data partitions from many temporal windows. In this section, we seek to answer the following questions: 1) When data partitions of different shuffling operations or even different queries are delivered to the same function instance, how to distinguish aggregation between data? 2) How to know the aggregation is complete and it's time to move on?

3.6.0.1 Payload Structure

The payload has a `Data` field that contains an "on-the-wire" representation of Arrow record batches. The `Schema` field defines the tables, fields, relationships, and types of the data carried. The payload also contains metadata about the data. For example, the `Encoding` field provides different compression options, such as Snappy [145], Lz4 [146] and Zstd [125], to compress Arrow data. The default option is Zstd.

```
Payload: { UUID, EpochID, ShuffleID, Data, Schema, Encoding }

UUID: { QID, SEQ_NUM, SEQ_LEN }

QID: <Query Code>-<Job ID>-<Query Timestamp>
```

To make shuffling and aggregation more deterministic, Flock marks each payload with a UUID. The `QID` copies the query code from the function name (see Section 3.5.4). Unlike the function name, the `QID` also contains the query start timestamp and job id. This allows different queries' payloads to be differentiated from one another. The `Epoch ID` indicates which microbatch the current data partition comes from. The payload's `UUID` also includes `SEQ_NUM` and `SEQ_LEN` in ad-

dition to the QID. SEQ_NUM is a monotonically increasing number used to identify the uniqueness of the payload in a certain set of aggregated data, and the SEQ_LEN field represents the total number of payloads needs to be aggregated. These two fields ensure the aggregate function knows whether all payloads have been collected for a certain job.

In the case of the partial aggregate inside the function, it produces multiple payloads, and each of them may be shuffling to different functions in the next function group (see Section 3.5.5). The Shuffle ID is used to assign an incremental number to each output payload in the function, and payloads (across function instances) in the same stage belonging to the same partition (range) are allocated the same shuffle id. This is mainly used to distinguish different aggregate tasks of the same query job since they can all be mapped the same next function. For example, the green payload's shuffle id is 1 and the blue payload is 2 in Figure 3.5.

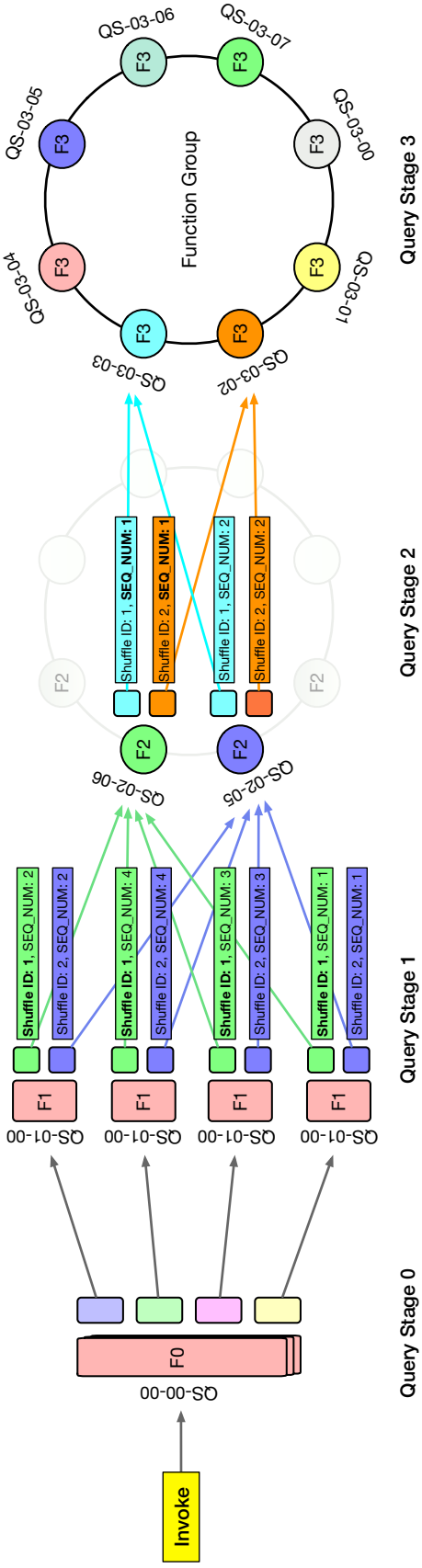


Figure 3.6: Multi-level Shuffling.

3.6.0.2 Global Arena

Static initialization happens before the query code starts running in the function. Listings 3.1 (line 23) shows the `INIT` code runs when a new execution environment is running for the first time, and also whenever a function scales up and the Lambda service is creating new environments for the function. The initialization code is not run again if an invocation takes effect on a warm instance. Static initialization is the best place to allow a function to reuse global resource in the same environment over multiple invocations. The cloud context and memory arena are deserialized or created in the initialization phase and are only loaded once per environment, avoiding them from being loaded on every invocation.

`Arena` is a global versioned hash map that aggregates data partitions outside of the function handler to ensure the data integrity for further stream processing. The key is a tuple including the `QID` and Shuffle ID of the payload. The value containing the currently received payloads. The `SEQ_LEN` field in each payload indicates the total number of payloads that the current session needs to collect. AWS Lambda does not yet provide per-instance concurrency [136], Flock decompresses and deserializes data only after receiving all payloads, allowing for maximum parallelization.

The function can receive the same payloads several times for async function invocation due to Lambda's internal queue is eventually consistent [116]. The bitmap field is provided for this reason: it guarantees that each payload is aggregated and processed only once. The payload's `SEQ_NUM` is a bitmap index that rep-

resents each payload as a single bit to track the aggregation state. The function can handle the same payload many times without incurring repeated duration expenses while utilizing bitmap. Even if the function output is empty, payloads carrying just metadata must be passed to the next function.

3.6.0.3 Multi-level Shuffling

Let's have a look at the query execution plan for an online auction system in Figure 3.2. The query is divided into four stages by Flock, and the whole execution flow on cloud functions is represented in Figure 3.6.

Stage 0: This stage reads upstream streaming data first, until the time window is reached. Separate cloud functions can be used for the auction and bid data sources, however just for simplicity, both data sources (`Bid` and `Auction`) are read in the same running instance. The repartition operator uses a hash of an expression (the join key) and the number of partitions (here, M=4) to map N input partitions to M output partitions. The data to be distributed in such a way that the same values of the keys end up in the same partition or payload. To deliver payloads to the next query stage, Flock calls the next function 4 times.

Stage 1: Lambda starts four instances, one for each payload to process. Each input payload has a distinct `SEQ_NUM` ranging from 1 to 4. The function then performed a local hash aggregation and repartitioned the partitions into two output payloads (green and blue boxes) after the hash join. The output payload inherits the input payload's `SEQ_NUM`. Based on the payload position, shuffle ids are as-

signed in increments of one. Each function uses the same deterministic seed to generate the same hash key, then does a single lookup to establish a starting point in the hash ring, then picks each next function counterclockwise for each payload and calls it in parallel.

Stage 2: The current function, unlike stage 1, collects multiple input payloads in the global arena (see Section 3.6.0.2). Shuffle ids are allocated in the same way in the current function's output payloads. The input's shuffle id, on the other hand, must be set to the output's `SEQ_NUM` so that the next function name can determine whether or not payloads are duplicates that can be aggregated by the same aggregate job.

Stage 3: The third stage produces output partitions, and its next function is a data sink action that delivers the result to downstream services in the current function.

3.7 Evaluation

In our evaluation we seek to answer the following questions in corresponding sections:

1. How does x86_64 and arm64 architecture affect Flock performance?
2. How performant is Flock compared to alternatives?
3. How does Flock's operational cost compare to alternatives as query workloads change?

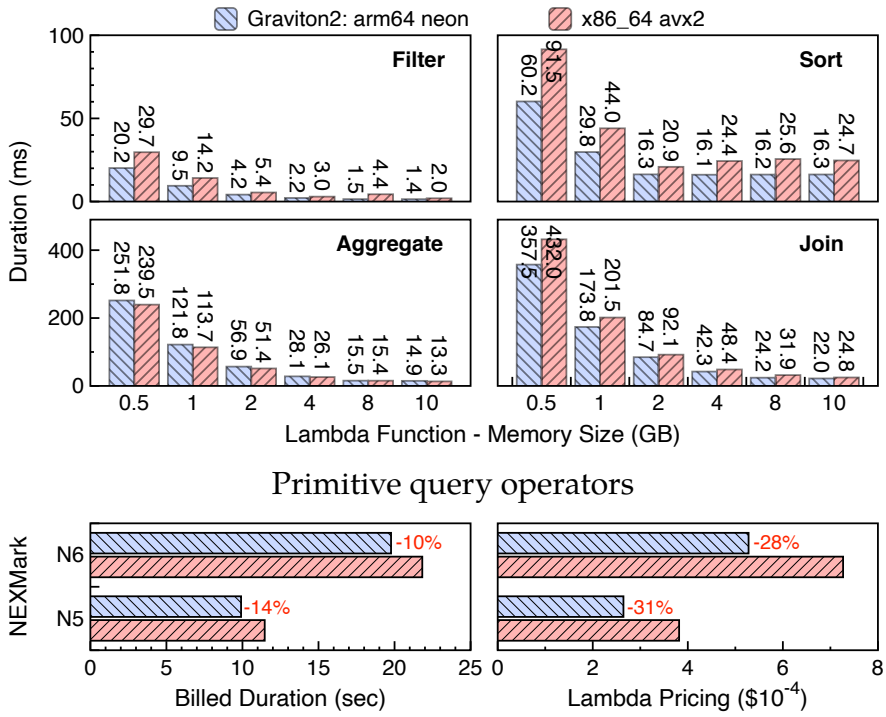


Figure 3.7: Lambda function on x86 and Arm processors.

3.7.1 Experimental Setup

To evaluate Flock's performance cost, we run our experiments on the following two streaming benchmarks: Yahoo Streaming Benchmark (YSB) [92] is a simple advertisement application, and its job is to read various JSON events from Kafka and store a windowed count of relevant events per ad campaign into Redis. NEXMark Benchmark [91] is an evolution of the XMark benchmark for an online auction house. NEXMark presents a schema of three concrete tables, and a set of queries to run in a streaming sense. NEXMark attempts to provide a benchmark that is both extensive in its use of operators, and close to a real-world application by being grounded in a well-known problem. The original benchmark was

adopted and extended by the Apache Foundation for their use in Beam [147], a system intended to provide a general API for a variety of streaming systems. To make things a bit more dynamic, they changed the size of the windows to be merely ten seconds, rather than the minutes and hours the original specification sets. They also added more queries [148], for example, q1 - q8 are from original NEXMark queries, q0 and q9 - q13 are from Apache Beam[6]. We follow the Beam implementation, as it is the most widely adopted one.

3.7.2 x86 vs Arm Architectures

The AVX2 and ARM Neon intrinsics are used in this experiment, which rely on Rust SIMD auto-vectorization as well as handwritten arrow kernels that explicitly employ SIMD intrinsics. We generated 500,000 NEXMark events, including 9995 person events, 29985 auction events, and 459770 bid events. Each subplot performs a different operation on the events. The proportion of a function's allotted memory determines the CPU share dedicated to it. This is why we tweak total memory to tune the CPU [149].

Figure 3.7(a) shows the performance of the lambda function executing fours query operators (filter, join, aggregate and sort) under the x86 and Arm architectures while varying the function's memory size. Except that Arm is 5-10% slower than x86 in aggregate operations, the billed duration of all other operations is less on Arm than on x86. The filter's duration accounts for 34% - 77% of total x86

[6]The NEXMark Query 13 is BOUNDED_SIDE_INPUT_JOIN: Joins a stream to a bounded side input, modeling basic stream enrichment.

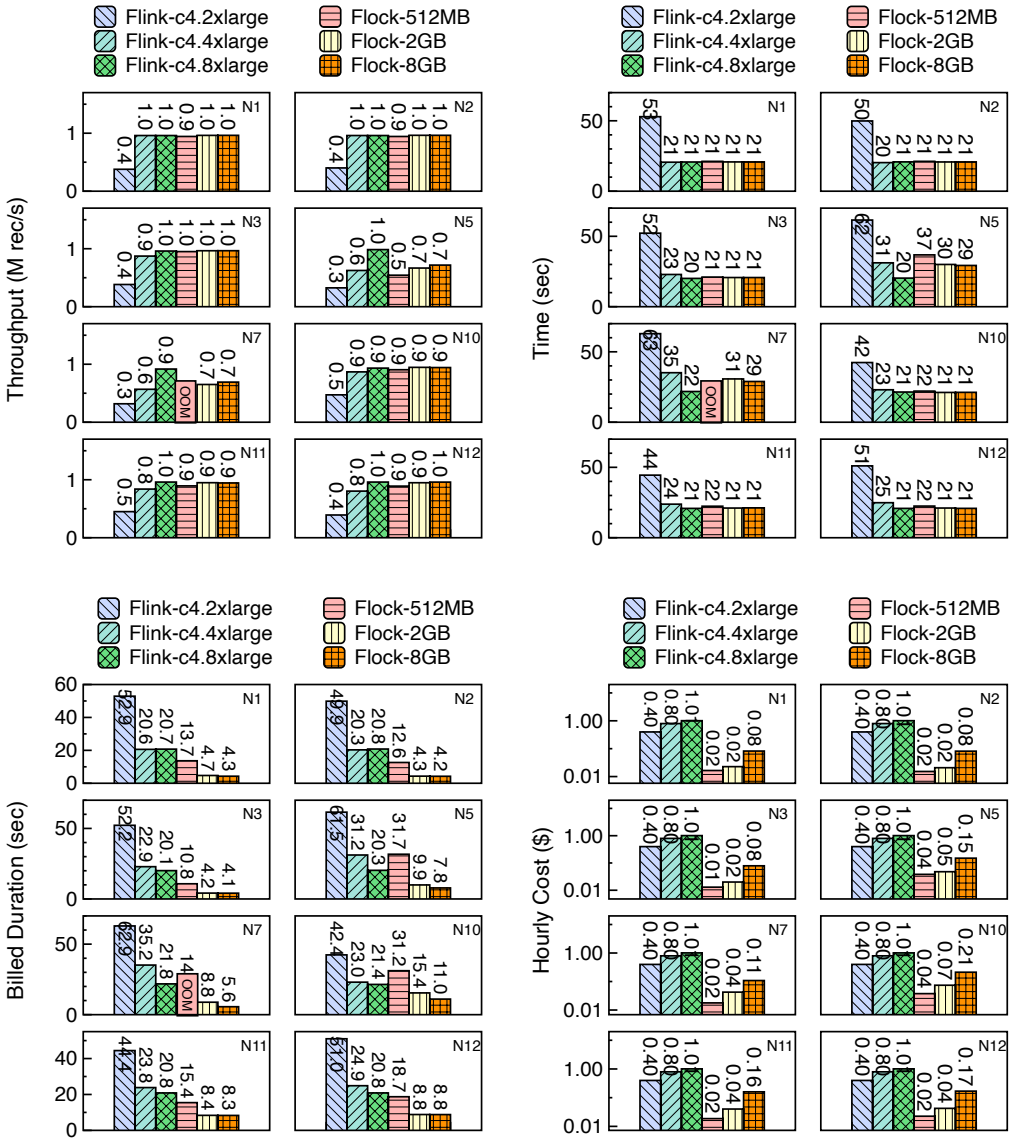


Figure 3.8: Performance cost of executing 20 million NEXMark events and 1 million events per second.

time, the join's duration for 76% - 91% of total x86 time, and the sort's duration for 61% - 76% of total x86 time on Arm. Furthermore, Arm's duration charge is 20% less expensive per millisecond than x86. For example, the 1ms charge for ARM 512MB is $0.0000000067, which is 20% cheaper than $0.0000000083 for x86 [150]. When compared to traditional x86 architecture on cloud, Flock on AWS Graviton2 processor saves more billing cost due to the shorter duration and lower charge.

Figure 3.7(b) shows the difference between NEXMark N5 and N6 under different architectures. N5 introduces the first use of windowing in NEXMark, and requires a sliding window to boot, which calculates the hot items in the last 10 seconds and update every 5 seconds. N6 is the only one in NEXMark that makes use of `partition by`. Both queries executed for 20 seconds, 1M event per second. N5 runs 14% faster than x86 on Arm, which is 31% cheaper than x86. Similarly, N6 is also 28% cheaper. In comparison to x86, Arm is indeed faster and less expensive. The rest of Flock experiments are run on Arm-based Graviton2 processors.

3.7.3 Performance Cost

Figure 3.8 compares the throughput, query time, billed duration and hourly cost of executing 10 million events and 1 million events per second between Flink and Flock under different configurations. Flink was deployed on EC2 instances – c4.2xlarge, c4.4xlarge and c4.8xlarge respectively with different numbers of CPU cores and memory sizes. EC2 instances are long-running, we set the duration of Flink to be the same as the query time. However, for Flock, the billed duration

refers specifically to the execution part of the cloud functions and does not include data preparation and transmission. We configured 3 memory sizes for Flock's cloud functions — 512 MB, 2 GB and 8 GB. Flink uses 8 workers, which equals to the concurrency of Flock function. Flock updates the state asynchronously to S3, whereas Flink updates the state to the local RocksDB [67]. To avoid the compaction overhead in EC2 instances, we only compared Flock and Flink in our experiments with the hashmap state backend enabled.

We ran the NEXMark queries 1, 2, 3, 5, 7, 10, 11, and 12 in the experiments. N1, N2, and N3 are elementwise queries that feed Flock a micro-batch of events per second. N5 is a sliding window query that schedules overlapping events that occurred in the last 10 seconds and updates every 5 seconds. N7 is a tumbling window query that aggregates events using distinct time-based windows that open and close at 10 second intervals. N10 is a query to log all occurrences to the file system – Flink saves output to the local file system, whereas Flock saves data to S3. N11 is a session window query that groups events for the same user that occur at similar times, while filtering out periods of time when no data is available. N12 is a tumbling window query with a 10 second interval dependent on processing time.

The c4.2xlarge has 8 vCPUs and 15.0 GiB of memory, but the performance is still far inferior to the Flock-512MB. This is because Flink is a Scala-based implementation, whereas Flock is a Rust-based high-performance query engine that includes SIMD and mimalloc [117] and is based on Arrow DataFusion [96]. When using the c4.4xlarge or c4.8xlarge, Flink can generally obtain similar throughput

and query time as Flock. The duration of Flock-512MB on N10 is greater than query time, this is because the processing of events from distinct mini-batches or windows can be separated, and then we can invoke new cloud functions to process any stacking events so that pipeline parallelism hides part of the duration delay, thereby shortening the query response time. The query time isn't lower than 20 seconds, since we produce 20 million events in total, and only process 1 million events per second. When Flock-512MB is operating on N7, an out of memory error is thrown. This is due to N7's need for 676 MB of RAM to collect, decompress, and deserialize data. Even if the function is not completed correctly, it will be charged for the time it took.

As illustrated in the hourly cost subgraph. Flock can reduce the hourly cost to 1/10 with similar performance to Flink. When the streaming data rate is low, the volume is modest, or the data is queried rarely, Flock's cost performance is more than two orders of magnitude better than Flink.

3.7.4 Invocation Payload

Table 3.2 shows the difference in latency of payload and S3 communication when Lambda memory size is set to 128MB, rather than end-to-end query processing. The coordinator overhead of the state-of-the-art system such as Starling [15], for example, is not included. Figure 3.9 therefore compares the invocation with payload to S3 communication in terms of latency, duration and billed cost while varying the number of events on NEXMark Q3. The memory size of the Lambda

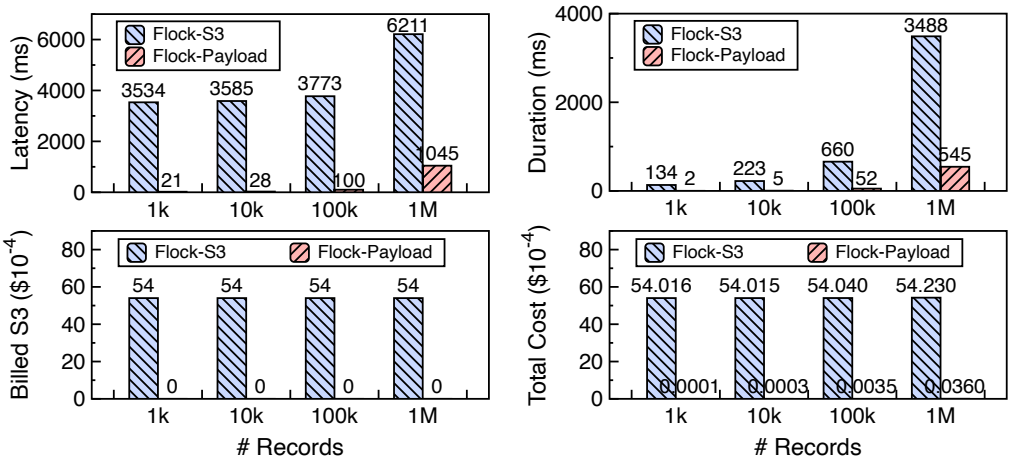


Figure 3.9: Flock Payload versus Flock S3 on NEXMark Q3.

function is set to 512MB in this experiment, and it is launched in `us-east-1`, with Flock-S3's coordinator is deployed on the client-side.

For Flock-S3, the latency minus the duration, which is about 3 seconds, is used to indicate the overhead of coordinator and function calls. Flock-S3 is an order of magnitude slower than Flock-Payload due to the round trips between the coordinator and cloud functions. It also has a one-order-of-magnitude higher billed duration cost than Flock-Payload. This is because S3 reads and writes all happen during function execution, and I/O latency is billed. In the case of Flock-Payload, however, raising the number of events increases the payload size, which impacts delivery time and execution time, and hence query latency, but delivery time has no effect on the billed duration.

The S3 subgraph shows the cost of utilizing S3 as an external communication medium for query processing. `PUT`, `COPY`, `POST`, `LIST` requests are charged $0.005 per 1,000 requests, and `GET`, `SELECT`, and all other requests are charged $0.0004 per 1,000 requests [151]. For S3 reads and writes, it actually means we are billed

`ceil(requests / 1000) * 0.0054` where `requests` is the number of that type of request we made during a monthly billing interval within one S3 region. Here, because the total number of S3 requests is less than 1000, we are directly charged $0.0054. Flock-Payload does not use S3 to transmit data between functions, there is no extra cost. The total cost is shown in the last subgraph, with the integer component coming from S3 communication and the fractional part coming from the duration cost.

3.7.5 Distributed Query Processing

When compared to query execution on a single function, distributed query execution has a substantial overhead and should be applied only when there is benefit in doing so [152]. Distributed execution splits the query plan into query stages, each of which is handled by a lambda function or function group. First, it can handle larger volumes of data. Due to each Lambda function can be allocated up to 10 GB of memory [88], it partitions input data into distinct function instances using hash shuffling, each of which performs the query independently. Furthermore, since AWS Lambda currently only supports the single request paradigm, the aggregate function must obtain all shuffled data from the previous query stage in a serial manner. Distributed execution can dramatically minimize latency by lowering the payload size and number of aggregates.

Figure 3.3 shows the latency and billed duration of NEXMark Query 4 and YSB under centralized and distributed executions. We produced 10 million events

for NEXMark N4. For centralized mode, Flock invoked the same function instance 65 times to complete N4. In distributed mode, ordinary lambda functions have a concurrency of 1000 by default , the aggregate function group has a size of 8, and each function member has a concurrency of 1. The latency of N4 is reduced by 4 times using distributed mode, but the billed duration is indeed 10 times that of the centralized mode. This is because N4 is divided into four query stages, each of which is called multiple times due to shuffle or aggregate, and each function execution results in a billable duration.

YSB models a simple ad account environment, where events describing ad views enter the system and those of a certain type are accounted to their associated campaign. Every ten seconds Flock is expected to report the total ads for each campaign within those ten seconds. YSB is a tumbling window of 10 seconds, and we generate 1 million events per second. Because ad event characteristics are all string types, and a single ad event is much larger than a NEXMark event, we raised the capacity of the Lambda function to 8GB. This is because the centralized mode cannot process queries in a 2GB memory environment. The centralized version of YSB has a substantially higher latency than the distributed mode by an order of magnitude. This is also because the ad event is excessively big, requiring Flock to invoke the same function instance 540 times in order to collect 10 seconds of window data and run the query. On the other hand, the billed duration in the distributed mode is comparable to that in the centralized mode, implying that distributed query processing has clear benefits on YSB.

Query	Mode	Memory	Latency	Billed Duration
N4	centralized	2G	17.49s	6.25s
N4	distributed	2G	4.12s	59.42s
YSB	centralized	8G	113.38s	31.54s
YSB	distributed	8G	2.95s	33.53s

Table 3.3: Distributed query processing.

3.7.6 Cold Start

Figure 3.10 shows the latency and billed duration while running NEXMark N3 multiple times with different number of events per second. A cold start is the first request that a new Lambda instance handles. This request takes longer to process because the Lambda service needs deploy our code and spin up a new microVM [140] before the request can begin. The first request handled by a Lambda instance will also trigger a one-time function that initializes the Lambda execution context from the cloud environment (see line 23 in Listings 3.1). When the number of events per second is 1K or 10K, both the first and second invocations to the lambda function have a long delay. The second call's billed duration has decreased dramatically, indicating that it is not a new instance, but its overall time has risen to 1.6 seconds. The third subgraph, on the other hand, only has one "cold start". We believe this is some unexplained behavior of AWS Lambda infrastructure. As expected, warm runs had a one- to two-order-of-magnitude decrease in latency. For stream processing, as long as the maximum idle time limit is not exceeded, Flock won't be troubled by cold run because Lambda is almost guaranteed to be warm since the query is executed continuously. For workloads that

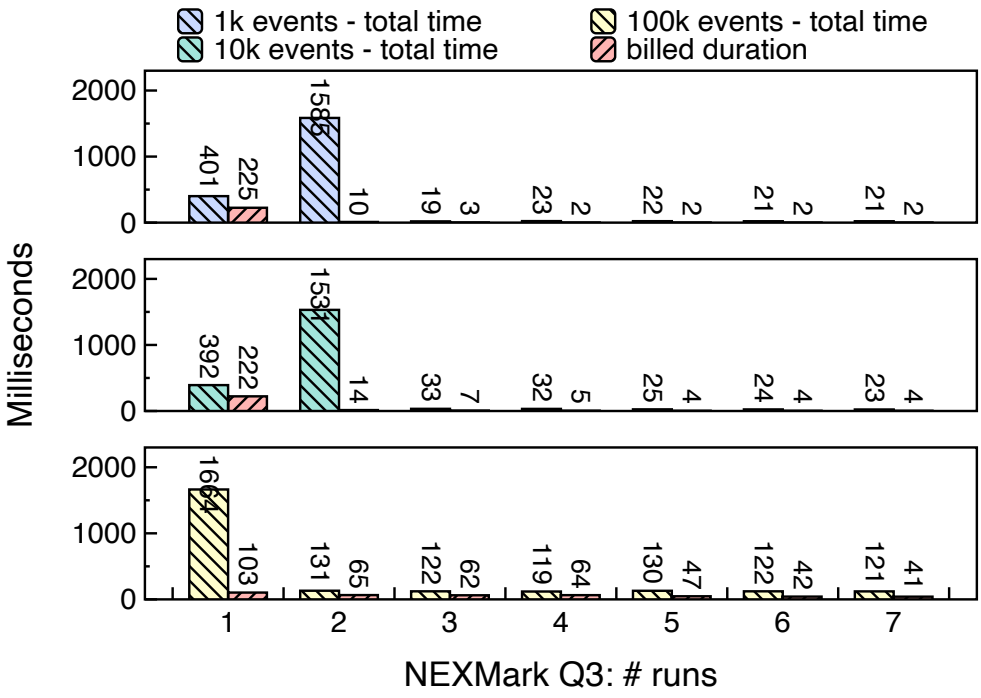


Figure 3.10: Lambda cold start cost on NEXMark Q3.

exceed the idle time limit, AWS Lambda supports provisioned concurrency [153] with extra cost that initializes a requested number of execution environments to respond immediately to the function's invocations.

3.8 RELATED WORK

Serverless workflows. Major cloud providers introduced serverless workflow services[154, 155, 156], which provide easier design and orchestration for serverless workflow applications. Netherite [157] and Kappa [158] are distributed execution engines that offers high-level language programming environment to execute Durable Functions efficiently. These frameworks are complete programming solutions that support advanced features (arbitrary composition, critical

sections), but they are not well-suited for supporting large, complex analytics jobs. Because they involve manually combining operators into a DAG utilizing vendor LOCK-in API. For example, Netflix's Conductor [159], Zeebe [160], and AWS Step Function [154] use a JSON schema for authoring workflows, and Fission Workflows [161], Google Cloud Composer [156], and Fn Flow [162], are somewhat more code-based, as the schema is constructed in code. Without query optimizer, the customized jobs are error-prone and suboptimal, resulting in significant performance loss, and are seldom reused for streaming workloads. Instead, Flock supports Dataframe and SQL API to make streaming computation more accessible to users.

Data passing is a key challenge for chained cloud functions. Pocket [131], Locus [13] and Caerus [16] implement multi-tier remote storage solutions to improve the performance and cost-efficiency of ephemeral data sharing in serverless jobs. Cloudburst [163] proposes using a cache on each lambda-hosting VM for fast retrieval of frequently accessed data in a remote key-value store, adding a modicum of statefulness to serverless workflows. Lambada [14] and Starling [15] use S3 as exchange operators for shuffling large amounts of data. SONIC [164] uses a hybrid and dynamic approach to choose data passing methods (VM- or Remote-Storage) automatically between any two serverless functions. Compared to state-of-the-practice systems, Flock is the first system to build a steaming query engine for data passing on cloud function services using the payload of function invocations, which is a general solution aimed at major cloud vendors without using external communication mediums.

Serverless streaming analytics. Flock shares the similar vision with Orleans on the virtual actor model [165, 166], an actor is automatically activated on demand to enable a serverless event-stream processing service with pay-for-use. The most similar work to ours may be Apache Flink Stateful Functions [167]. Flink takes care of the state and messaging, while the application runs as a stateless Kubernetes deployment, or as FaaS functions. Flink's TaskManagers are coordinators that manage the state, handle the messaging, invoke the stateful functions and go through a service that routes the resulting messages to the next respective target functions, for example a Kubernetes (load-balancing) service, the AWS request gateway for Lambda, etc. Stateful Functions, which are atomic units of isolation, distribution, and durability, are now the building blocks of applications that are similar to Azure Durable Functions. Flock, on the other hand, is the first cloud-native streaming processing engine to enable SQL on FaaS with heterogeneous hardware (x86 and ARM) with the ability to shuffle and aggregate data without the need for a centralized coordinator or remote storage like S3.

Chapter 4: BullFrog: Online Schema Evolution via Lazy Evaluation

BullFrog is a relational DBMS that supports single-step schema migrations — even those that are backwards incompatible — without downtime, and without need for advanced warning. When a schema migration is submitted, BullFrog initiates a logical switch to the new schema, but physically migrates affected data lazily, as it is accessed by incoming transactions. BullFrog's internal concurrency control algorithms and data structures enable concurrent processing of schema migration operations with post-migration transactions, while ensuring exactly-once migration of all old data into the physical layout required by the new schema. BullFrog is implemented as an open source extension to PostgreSQL. Experiments using this prototype over a TPC-C based workload (supplemented to include schema migrations) show that BullFrog can achieve zero-downtime migration to non-trivial new schemas with near-invisible impact on transaction throughput and latency.

4.1 Introduction

Continuous Deployment (CD), an aspect of DevOps [168], is the increasingly popular practice of frequent, automated deployment of software changes [169, 170], with some practitioners deploying multiple changes per day [171]. To realize the benefits of CD, it must be straightforward to deploy updates to both front-end code and the database, *even when the database's schema has changed*. Unfortunately, this is where current practices run into difficulty. [171] stated in their retrospective on CD practices at OANDA, "database schema changes always were always ad hoc and full of fear." [169] and [170] confirmed via surveys of more than 100 experts and practitioners that database schema changes are particularly challenging to handle. Nonetheless, there is evidence that schema changes are also frequent: [18] examined changes in releases to a dozen open-source applications and found that schema changes occurred roughly once per week, on average. In our own examination of the development history of 20 open source Ruby on Rails applications found on GitHub (including the Sharetribe and GitLab repositories studied by Bailis et al [172]) we found 1,611 schema changes, of which approximately 20% required significant physical data movement.

Application developers should be free to change the code and database schema as they see fit, without concern for the complexities of deploying those changes later. For example, they should be able to delete or add columns or constraints, join tables, split tables, etc. according to their needs. To deploy an update, the developer defines an *evolution transaction* that is used to migrate the existing data

to the new schema. The CD system uses that transaction to update the current database, and then deploy the updated front-end instances. If downtime is not a concern, then the simplest way to do this is to shut down all of the application instances, migrate the data, and then restart with the new instances. Of course, downtime frequently *is* a concern, and with multiple updates happening per day, the simple shutdown-and-restart approach is unacceptable.

State-of-the-art approaches to schema migration without downtime use a **multi-step** approach. In such approaches, the database system migrates a copy of the data to the new schema in the background, before the front-end instances can switch over to it. During this migration window, writes to the old schema must be propagated to the new copy, typically via triggers or log shipping of updates [173, 174, 175, 176, 177, 178, 179, 180]. Aside from the obvious side effect that such approaches approximately double the resource requirements of the system, the requirement to **delay** the front-end application migration until the database is ready is fundamentally antithetical to the spirit of the continuous deployment movement.

To avoid such delays, new-version transactions can be rewritten so they work on the old schema while the data is migrated; rewriting can occur in the other direction too, to allow both application versions to coexist [173, 181, 182, 183, 184, 185]. Unfortunately this limits schema evolution to backwards compatible migrations, since transactions must be rewritable/processable over all active schema versions. For example, if an application evolves such that data must be inserted that violates previously defined integrity constraints, the constraints cannot sim-

ply be dropped in the new schema, because doing so would be backwards-incompatible: this data cannot be inserted into the old schema. To cope, developers may be forced to employ non-ideal structures and/or temporary tables and front-end code, thereby accruing technical debt in applications and *decay* in databases [186], while adding complexity to the update process (see the OANDA quote above). All of these problems have pushed application developers toward "NoSQL" databases that avoid predefined schemas entirely, and never reject writes that use a different schema than has been previously used.

In this chapter we propose a mechanism for schema evolution, that we call BullFrog[1], that avoids the delayed deployment of the new schema required in the multi-step approaches, while also avoiding the migration restrictions and database decay of the backwards compatible approaches. BullFrog deploys **arbitrary** schema changes immediately, in a **single step**, in conjunction with CD-based updates to a web-based or mobile service. Rather than introduce downtime by halting existing service while the evolution transaction takes place, BullFrog *logically* converts the database to use the new schema immediately without any *physical* changes of the stored data. Updated front-end instances may now submit transactions using the new schema. Doing so prompts BullFrog to migrate any relevant tuples from the old schema's tables to the new/updated ones before processing the transaction; i.e., tuples are migrated *lazily*, as needed. For backward-compatible schema changes, BullFrog permits old-version front-end instances to be updated gradually; for incompatible changes, front-end instances are updated as a *big flip* [187];

[1]Bullfrogs do not sleep, much like our zero-downtime migration system.

the latter can be done by simply restarting them (e.g., when they submit an incompatible query [188]), or using a more sophisticated dynamic software updating scheme [189, 190, 191].

BullFrog uses a light-weight concurrency control mechanism that ensures exactly-once migration of data under contention. Shared data structures synchronize record migration states and allow database system workers that are processing separate transactions to cooperate in parallel without missing or duplicating tuples, and without landing in stuck states due to aborts or cyclic dependencies.

We have implemented a prototype of BullFrog as an extension to PostgreSQL. We use this prototype to evaluate BullFrog by measuring its performance on variations of the standard TPC-C benchmark that include schema migration transactions. We find that under realistic deployment scenarios, BullFrog is able to support complex schema migrations in a near invisible fashion, with no downtime, no observable effects on system throughput, and limited latency increases. BullFrog is thus the first system, to the best of our knowledge, to support single-step, non-backwards compatible schema migrations without downtime.

In summary, BullFrog makes the following contributions:

- A proposed system design (BullFrog) that uses *lazy schema migration* to implement single-step, on-line schema evolution; our proposal is that schema migrations are installed quickly and data is migrated on demand as queries are serviced subsequently. This approach has the benefit of reducing overall downtime.

- Algorithms and data structures that achieve efficient, exactly-once physical migration of data under contention.

- An implementation of BullFrog in PostgreSQL supports three major categories of schema migration: projections, aggregations and joins; they can be used as primitives for expressing other schema migrations.

- An extension of the TPC-C benchmark that includes non-trivial and non-backwards compatible schema migrations.Our experiments on this benchmark demonstrate that our technique can reduce query latencies w.r.t the eager migration techniques by more than an order of magnitude while maintaining acceptable transactional throughput during schema migration.

4.2 Request-Driven Lazy Migration

Many modern database systems, however, allow the schema evolution to be performed online. Such migrations are mostly restricted to simple schema changes such as changing the name of a column, adding an index, etc. and does not handle non-trivial schema changes. We focus on particularly those schema changes that also results in data movement as a part of the schema change operation. These schema changes take longer to execute with respect to the former schema changes since it involves movement of data from one table to another. Typical examples of such schema changes include adding a new column to a table with data values obtained from other tables, selecting only certain columns or joining two tables to form a new table, splitting a table into two tables, storing a

table aggregation into another table.

In particular, we propose BullFrog, a system that performs online schema evolution, where the old schema is migrated lazily to the new schema.

4.2.1 Basic approach

A schema migration request is submitted to BullFrog as one or more DDL statements. These statements may create new tables or modify or delete existing tables. Data from existing tables may be specified to initialize or update new or modified tables. The new schema becomes immediately active as soon as it is processed by BullFrog. For non backwards-compatible "big flip" migrations, the old schema becomes inactive, and all subsequent requests that access it are rejected. For requests over the new schema, BullFrog identifies tuples in the old tables that are potentially relevant, physically migrates them to the new/modified tables, and then processes the original request on the new/modified tables.

A key question is how to determine which old-schema tuples are "potentially relevant" and should thus be moved to the new tables. In BullFrog this is done in two steps. First, when the schema change is first installed, a read-only VIEW is set up between the old and new tables based on the migration statements submitted by the client. Then, when a user submits a query or transaction against the new schema, BullFrog uses any predicates that are found in the query to essentially refine this VIEW, i.e., to limit the set of tuples still located in the old table(s) that are potentially relevant to the query result.

Once the relevant tuples are identified any required data movement can take place (if all data is already present, the transaction can go ahead immediately). This is done by (1) using the VIEW to convert queries over the new schema into queries over the old schema using standard database VIEW expansion; (2) extracting clauses from the query plan over the old schema that restrict the set of tuples that are accessed; (3) inserting these self-same restrictive clauses into the previously submitted schema migration DDL;[2] and (4) processing this subset of the migration operation and inserting the results into the new tables.

BullFrog ensures that data is migrated from the old to new tables exactly once, even in the presence of concurrent transactions. Indeed, concurrent transactions can accelerate the migration process by simultaneously operating on different parts of the database, or different parts of the same tables. We discuss the concurrency module in detail in Section 4.3; the remainder of this section focuses on transaction rewriting as if there is but a single (sequential) client.

Consider a hypothetical airline flight application with an original schema containing two tables:

```
CREATE TABLE FLIGHTS (FLIGHTID CHAR(6) PRIMARY KEY, SOURCE
  CHAR(3), DEST CHAR(3), AIRLINEID CHAR(2), DEPARTURE_TIME
  TIMESTAMP, ARRIVAL_TIME TIMESTAMP, CAPACITY INT);
CREATE TABLE FLEWON (FLIGHTID CHAR(6), FLIGHTDATE DATE,
  PASSENGER_COUNT INT CHECK (PASSENGER_COUNT > 0));
```

The `FLIGHTS` table contains general information about active flight routes, and the `FLEWON` table contains daily flight statistics.

[2] Since the new table has been created already, the original DDL statement is changed to a DML statement.

At some point the application developer makes some schema changes: (i) rename FLEWON to FLEWONINFO; (ii) add a derived attribute, EMPTY_SEATS; (iii) add attributes ACTUAL_DEPARTURE_TIME and ACTUAL_ARRIVAL_TIME to track the delay incurred by each flight; and (iv) drop constraint (PASSENGER_COUNT > 0) to allow for the airline to take packages rather than passengers during a pandemic. Change (iv) is backward-incompatible.

This change is expressed with the following migration DDL:

```
CREATE TABLE FLEWONINFO AS (
  SELECT F.FLIGHTID AS FID, FLIGHTDATE, PASSENGER_COUNT,
         (CAPACITY - PASSENGER_COUNT) AS EMPTY_SEATS,
         DEPARTURE_TIME AS EXPECTED_DEPARTURE_TIME,
         NULL AS ACTUAL_DEPARTURE_TIME,
         ARRIVAL_TIME AS EXPECTED_ARRIVAL_TIME,
         NULL AS ACTUAL_ARRIVAL_TIME
  FROM   FLIGHTS F, FLEWON FI
  WHERE  F.FLIGHTID = FI.FLIGHTID);
```

Once the migration has been submitted, BullFrog creates a new transaction that creates new, empty tables corresponding to the tables that are created or modified in the migration request, along with a temporary VIEW (that will only be used during the migration process) that contains the contents of the migration request:

```
CREATE VIEW FLEWONINFO_VIEW AS (
  SELECT F.FLIGHTID AS FID, FLIGHTDATE, PASSENGER_COUNT,
         ...); -- exactly matches FLEWONINFO def above
```

BullFrog also creates data structures (described in Section 4.3) to control concurrent migration processes and track migration status.

When the database system receives a client request that references any table that was added/modified, BullFrog first migrates any relevant data from the old tables, and then processes the original request over the new ones. To do this, it uses filtering statements in the client request, typically located in the WHERE clause of SELECT, UPDATE, and DELETE statements, to limit the scope of the lazy migration. BullFrog attempts to convert these filters over the new schema into filters over the old schema that match as few tuples as possible while still yielding the set needed to fully process the client request.

As an example, consider the following client request.

```
SELECT * FROM FLEWONINFO WHERE FID = 'AA101'
    AND EXTRACT(DAY FROM FLIGHTDATE) = 9;
```

The `FID = 'AA101'` predicate will be converted into `FLIGHTID = 'AA101'` over the `FLIGHTS` and `FLEWON` tables, while the predicate `EXTRACT(DAY FROM FLIGHTDATE) = 9` will be applied on the `FLEWON` table. Only those tuples from these old tables that return `true` for these converted predicates need to be migrated; all other tuples can be ignored and migrated at a later time.

Although for this example it was straightforward to convert the predicates over the new schema into predicates over the old schema, in some cases such a conversion is non-trivial or impossible. In the worst case, all tuples in the old schema must be deemed potentially relevant (see Section 4.2.4).

BullFrog uses the `VIEW` that it created during the migration initialization step discussed above to leverage existing capabilities in database systems to move filters across schemas. Most database systems implement *view expansion* to rewrite

requests over a view into requests over the original tables. BullFrog accesses the query plan that was generated after view expansion and query optimization, and extracts any filtering statements over the old schema.

For the example client request, the output from PostgreSQL `EXPLAIN` (which shows its query plan) is shown below:

```
                              QUERY PLAN
-----------------------------------------------------------------
Nested Loop  (cost=4.34..14.04 rows=1 width=99)
->  Seq Scan on flights f  (cost=0.00..4.50 rows=1 width=27)
    Filter: (flightid = 'AA101'::bpchar)
->  Bitmap Heap Scan on flewon fi
    (cost=4.34..9.52 rows=1 width=15)
    Recheck Cond: (flightid = 'AA101'::bpchar)
    Filter: (date_part('day'::text, (flightdate)::timestamp
    without time zone) = '9'::double precision)
    ->  Bitmap Index Scan on flewon_flightid_idx
    (cost=0.00..4.34 rows=9 width=0)
            Index Cond: (flightid = 'AA101'::bpchar)
```

The predicates over the view have been converted into predicates over the original tables; we see the predicate `FLIGHTID = 'AA101'` on both `FLIGHTS` and `FLEWON` table (line 5 and 13 in the query plan) and the predicate `EXTRACT(DAY FROM FLIGHTDATE) = 9` on the `FLEWON` table (line 9-10 in the query plan). BullFrog inserts these predicates into a version of the original schema migration DDL, except that the `CREATE TABLE` statement in the query is substituted with an `INSERT INTO` statement, as shown below.

```
INSERT INTO FLEWONINFO (
  FID, FLIGHTDATE, PASSENGER_COUNT, EMPTY_SEATS,
  EXPECTED_DEPARTURE_TIME, ACTUAL_DEPARTURE_TIME,
```

```
  EXPECTED_ARRIVAL_TIME, ACTUAL_ARRIVAL_TIME)
(SELECT F.FLIGHTID, FLIGHTDATE, PASSENGER_COUNT,
        (CAPACITY - PASSENGER_COUNT),
        DEPARTURE_TIME, NULL, ARRIVAL_TIME, NULL
 FROM   FLIGHTS F, FLEWON FI
 WHERE  F.FLIGHTID = FI.FLIGHTID
 AND    F.FLIGHTID = 'AA101' AND FI.FLIGHTID = 'AA101'
 AND    EXTRACT(DAY FROM FLIGHTDATE) = 9);
```

BullFrog implements DELETE and UPDATE statements by rewriting them into SELECT statements on the old schema to migrate relevant tuples first, and then processing the original request on the new schema. This limits BullFrog's reliance on views to read-only queries for which view expansion is trivial (via nesting SQL statements in the FROM clause), and avoids the well-known problem of performing updates through views.

INSERT commands generally can be performed over the new schema without requiring any prior migration unless there are integrity constraints defined on the new schema. Such constraints may expand the set of potentially relevant data beyond the data specified by the client request. For example, if a uniqueness constraint is defined on any column of the new table, then any INSERT commands over the new schema (or updates to the unique attribute) must first migrate records that have potentially conflicting values so that the constraint can be properly checked over the new schema.

4.2.2 Background migrations

If parts of the input tables are never deemed relevant for client requests, a purely lazy system will never migrate them. To ensure that all data is eventually migrated, BullFrog initiates background migration threads that slowly inject simulated client requests that cumulatively cover the entirety of the old tables. When these threads finish, the migration is complete and the old schema can be deleted.

4.2.3 Consistency

Unlike many of the state-of-the-art schema migration approaches discussed in Section 4.1, BullFrog does not maintain replicas as part of its approach. Logically, a given tuple starts in the old schema and eventually migrates to the new schema, but never exists in both schemas simultaneously. Once it is migrated, although a stale physical copy may remain in the old schema, the migration status of that tuple prevents it from subsequently being accessed. Thus, there is no concern for replica consistency in BullFrog[3].

With regard to ACID consistency, BullFrog does not restrict what constraints may be declared on the old schema or new schema. However, it does not automatically generate integrity constraints based on the integrity constraints that existed in the old schema. Rather, the schema migration DDL must explicitly (re)declare any integrity constraints that must be enforced on the new schema. We will analyze the performance impact of needing to migrate coarser units of data in order

[3]Such as the notion of consistency used by the CAP [192] and PACELC [193] theorems.

to check integrity constraints in Section 4.4.5.

4.2.4 Limitations

BullFrog supports any legal SQL that can appear in DDL statements in the database system, including any legally defined integrity constraint over the new schema. However, some types of migrations reduce BullFrog's effectiveness.

First, although BullFrog utilizes views in a read-only fashion, (thereby avoiding the view update problem), any limitations in the view support of the underlying database system is passed through to BullFrog. One situation where this limitation is manifested is when the migration is expressed using a user-defined function (UDF) instead of standard SQL. Many database systems support the incorporation of calls to UDFs within views, but treat the UDF as a black box during query planning. Any filtering conditions that appear within the UDF code will be invisible to BullFrog and therefore will not be helpful to limit the scope of the lazy migration.

Second, integrity constraints added during migration may cause arbitrary data to be dropped. For example, if a uniqueness constraint is added to a table with duplicates, most existing systems will return an error immediately upon the ALTER TABLE command that attempted to add the constraint. However, a pure lazy migration approach would prevent the system from becoming aware of the problem until after the new schema is already live. Therefore, BullFrog must either perform a synchronous check upon receiving a potentially problematic mi-

gration command so that it can return an error in advance, or otherwise proceed with the pure lazy approach and give a warning that some records may fail to migrate.

4.3 Lazy Migration, Concurrently

BullFrog must support concurrent client requests that may access overlapping sets of data. Care must be taken to avoid migrating the same tuple more than once, or deleting it from the old tables prematurely. BullFrog addresses these problems by using custom data structures and mechanisms to track the status of a tuple as it is migrated. At a high level, the technique involves locking ranges of data in the old tables to ensure that once one worker begins to migrate the data, no other concurrent worker can attempt to migrate it unless the first worker fails. Care is taken to handle situations in which there is not a one-to-one mapping of tuples under the old schema to tuples under the new schema. Furthermore, efficient data structures are used to track the status of these locks and the history of previously migrated data. In sum, BullFrog's design allows conflicting migration efforts to continuously and efficiently make progress migrating non-conflicting records, and avoid duplicating work for conflicting records.

4.3.1 Migration categories

A schema migration may involve one or more migration statements. Each migration statement may involve one or more tables from the old schema ("input

tables") and generate one or more tables in the new schema ("output tables").[4] For each input table in a migration statement, BullFrog classifies it into four broad categories that dictate how its tuples will be locked and tracked during migration.

One-to-one (1:1) migration. In a 1:1 migration, each tuple in an input table has at most one corresponding tuple in the output schema (across all the output tables). Examples of 1:1 migrations include adding one or more columns to a table, dropping one or more columns from a table, changing the data type of a column, adding constraints to a table (which may cause the output table to be a subset of the tuples in the input table), or joining a table to another one using one of its foreign keys, i.e., a foreign-key, primary-key (FK-PK) join. For 1:1 migrations, BullFrog uses a bitmap to track migration and lock status. There are two bits per tuple in the input table: one bit corresponding to that tuple's migration status, and the other corresponding to its lock status. Bitmaps provide a favorable space-time trade-off, since they are effective at exploiting bit-level parallelism in hardware and introduce limited overhead. Tuple-level granularity on migration and lock status gives BullFrog the flexibility to migrate exactly those tuples that it has determined to be potentially relevant to a particular client request without having to drag along unnecessary data during the migration process. However, BullFrog also provides the capability to track migration and lock status at less granular levels (e.g. at a page level).

One-to-many (1:n) migration. In a 1:n migration, each tuple in the input table may (but does not necessarily) produce more than one tuple in the output

[4] We ignore migrations that involve zero input or output tables since they are trivial.

schema. One example of such a migration is where an input table is split into multiple output tables, with a single input tuple generating a tuple in each of the output tables. Other examples include the primary key side of a FK-PK join and either side of a many-to-many join. 1:n migrations work similarly to 1:1 migrations in that a bitmap is used to track migration and lock status. The only additional detail is that the migration bit for a tuple in the input table cannot be set (indicating that it has been migrated) until all of its dependent tuples in the output schema have been generated.

Many-to-one (n:1) migration. In a n:1 migration, a group of tuples from the same input table combine to generate a single tuple in the output schema. An example of an n:1 migration is where an output table is formed by performing a group-by aggregation. For these migrations, BullFrog tracks migration and lock status at the group level, and uses a hash table instead of a bitmap.

Many-to-many (n:n) migration. n:n migrations are implemented as an extension of n:1 migrations analogous to how 1:n migrations extend 1:1 migrations as described above. Thus, a hash table is used instead of a bitmap, but the migration bit for a group in the input table is only set once all of the group's dependent tuples in the output schema have been generated.

We call 1:1 and 1:n migrations "bitmap migrations", and n:1 and n:n migrations "hashmap migrations", and discuss each of these in more detail in the following subsections.

When the same input table is involved in separate migration statements, BullFrog maintains multiple data structures for it. For example, if a column is

Algorithm 1: Per-transaction migration loop.

```
1  do
2  |  WIP, SKIP = empty list
3  |  Start transaction
4  |  Scan via client request predicates, for each tuple, T do
5  |  |  canMigrate = Call Algorithm 2 or 3 for T
6  |  |  if canMigrate == true then  Include T in migration.
7  |  End transaction
8  |  for each tuple or group, G in WIP do
9  |  |  Update G's status to migrated (not in-progress)
10 while SKIP is not empty
11 Run client request on new schema
```

added to a table (1:1 migration) in addition to it generating a new table via a 1:n join (1:n migration), two bitmaps are allocated to manage the two migration operations on that table.

4.3.2 Migration transaction processing

The migration work precipitated by a client request is performed in a series of transactions that is separate from, and completed prior to, processing the client request transaction. Dividing work into multiple transactions simplifies abort handling and avoids deadlock.

Algorithm 1 shows BullFrog's per-worker logic for handling a client transaction during schema migration. Two worker-local lists, `SKIP` and `WIP`, start off empty (line 2). After starting a transaction, it iterates through the records in the old schema deemed to be potentially relevant via the predicate extraction process described in Section 4.2.1 (lines 3-4). For each relevant record, T is migrated if Algorithm 2 (bitmap migrations, Section 4.3.3) or Algorithm 3 (hashmap migra-

tions, Section 4.3.4) says it should be (lines 5-6).

Algorithms 2 and 3 add to `WIP` tuples or groups for which they returned *true*, and add to `SKIP` those for which they returned *false* due to an existing migration effort by a different worker. After the migration transaction completes, the status of all tuples in `WIP` are updated to indicate that they have been migrated (lines 8-9) using the data structures to be discussed shortly. Finally, the loop body ends and the `SKIP` list is checked (line 10). If it is non-empty, the **do** loop repeats (in a fresh transaction) to recheck the status of these skipped tuples and migrate them in the rare case that the other worker that was migrating them aborted (see Section 4.3.5).

4.3.3 Bitmap migrations

For 1:1 and 1:n migrations, lock and migration status are tracked using two bits per migration granule, such as a tuple or page. (This section assumes tuple granularity for simplicity.)

- A **migrate bit** that is initialized to 0 and is set to 1 when that tuple has been migrated.
- A **lock bit** (or "in progress" bit) that is initialized to 0 and is set to 1 when a worker begins the process of migrating this tuple. Setting the bit to 1 prevents other migration workers from concurrently trying to migrate the same tuple.

These two bits are stored in adjacent positions in the bitmap so both can be ac-

cessed in a single read of a memory word. The top of Figure 4.1 shows an example bitmap, with [*lock-bit migrate-bit*] pairs associated with a set of 8 tuples. A pair of [0 0] in the bitmap indicates that the tuple has not yet started the migration process, [1 0] means that the migration is "in-progress", and [0 1] means the migration has completed. A state of [1 1] should never occur.

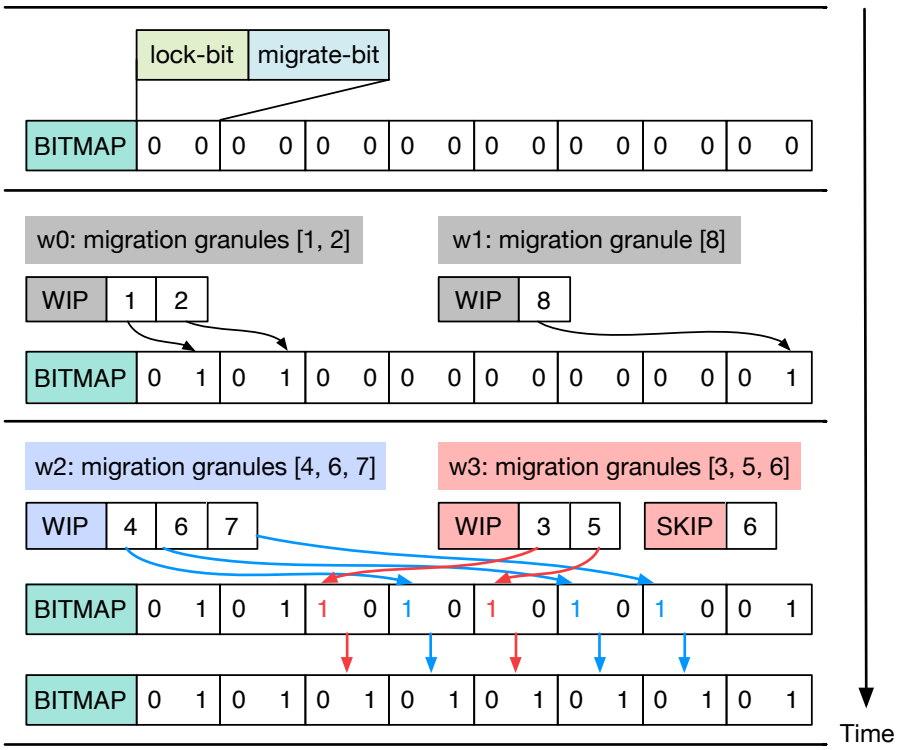


Figure 4.1: Schema migration during transaction processing.

The bitmap is protected from concurrent workers by a read-write latch that allows concurrent reads but requires exclusive access for writes. We partition the bitmap into separate chunks protected by different latches to reduce cross-worker latch contention.

To check whether to migrate a particular tuple, the worker runs the pseudocode shown in Algorithm 2. Line 1 checks the migration bit of the tuple. If it is

Algorithm 2: Check bitmap whether to migrate a given tuple.

Input : shared bitmap `bitmap`, tuple index in bitmap `bkey`, local in-progress lists `WIP` and `SKIP`
Output: true, if the worker is permitted to migrate the tuple

```
1  if migrate bit of bkey in bitmap is not set then
2      if lock bit of bkey in bitmap is set then
3          append bkey to list SKIP
4          return false
5      acquire an exclusive latch on the bitmap
6      if migrate bit of bkey in bitmap is not set then
7          if lock bit of bkey in bitmap is not set then
8              set the lock bit of bkey in bitmap
9              release the latch on bitmap
10             append bkey to list WIP
11             return true
12         else
13             release the latch on bitmap
14             append bkey to list SKIP
15             return false
16     else release the latch on bitmap
17 return false
```

1, it has already been migrated, so it returns false (line 17). Line 2 checks the lock bit. If it is 0, the code in lines 5-16 is run, that sets the lock bit to 1 and appends the tuple identifier into the in-progress list `WIP` of that worker. All of this is done after getting an exclusive latch on the bitmap partition and then rechecking the migration and lock bits to confirm they were not changed before the worker acquired the exclusive latch. If the lock bit is 1, another worker has already started the process of migrating this tuple. In that case, the code in lines 3-4 and 13-15 adds that tuple to the"skipped tuple" list `SKIP` for that worker.

Algorithm 1 migrates the tuple and at line 9 sets its status: [0 1].

The example in Figure 4.1 depicts four queries issued over the new schema, each of which spins up a worker to migrate relevant data (query `q1` spins up

worker `w1`, query `q2` spins up worker `w2`, etc.). w0 executes simultaneously with w1, but they do not attempt to migrate the same data, and so can proceed independently. After this, w2 and w3 run concurrently and both attempt to read data located inside the 6th tuple. Although reads do not conflict with respect to data access, they do conflict with respect to migration workers, since only one worker is allowed to migrate this tuple. w2 acquires the lock bit on tuple 6 earlier than w3, so w2 appends tuple 6 to its `WIP`, while w3 observes that tuple 6 is locked and so appends it to its `SKIP`. When w3 finishes migrating tuples 3 and 5, it checks tuple 6 again to see if it was migrated. If it was, it allows q3 to run on the new schema. Otherwise, it blocks the query until tuple 6 is migrated or the lock is released.

4.3.4 Hashmap migrations

Both n:1 and n:n migrations require accessing multiple tuples from an input table in order to produce an output tuple. This means that tuple-level granularity tracking of migration status is inappropriate—either an entire group of tuples that combine to form an output tuple should be considered migrated, or none of them. BullFrog therefore tracks lock/migrate status at the group level for these migrations. Since mapping arbitrary group identifiers to unique offsets in a dense bitmap would be complex without advanced knowledge of the complete set of group identifiers, a hash table is used to track statuses, rather than a bitmap.

Given a tuple in an input table that is potentially relevant to a query result, Algorithm 3, line 1, determines the group identifier to use as a key into the hash

Algorithm 3: Tuple eligibility checking for hash migrations.

```
   Input  : shared hash table htable and a tuple T,
            local in-progress lists WIP and SKIP
   Output: true, if the txn is permitted to migrate T
 1 Generate group key hkey from T
 2 if hkey exists in list WIP then return true
 3 if hkey exists in list SKIP then return false
 4 if hkey exists in htable then
 5 |   if hkey state in htable is in-progress then
 6 |   |   append hkey into list SKIP and return false
 7 |   if hkey state in htable is abort then
 8 |   |   update the pair (hkey, in-progress) in htable
 9 |   |   append hkey into list WIP and return true
10 |   return false
11 if htable.insert (hkey, in-progress) already exists then
12 |   GOTO LINE 7
13 else append hkey into list WIP and return true
```

table. For example, for a `GROUP BY` migration statement, the group identifier is constructed from the value of the attribute(s) that appear in the `GROUP BY` clause. With this key, the worker executes the remaining code in Algorithm 3.

Line 2: If the key exists in the list `WIP`, this means that this same worker has already decided to migrate a different tuple from the same group. Since the entire group must be migrated together, the worker must migrate the current tuple as well.

Line 3: If the key is found in the list `SKIP`, a different worker was already found to be migrating the group associated with this tuple and so it will be skipped by the current worker and revisited a later point to check whether the other worker successfully completed the migration of this group, as we described in Section 4.3.2.

Lines 4-10: If the key is found in the global hash table (but not the local lists),

its current lock/migration status is checked. If it is locked but not yet migrated, this implies that the migration is `in-progress` by another worker, and the key is appended to the local list `SKIP`. If it is neither locked nor migrated, this implies that a different worker started the process of migrating it, but aborted. The worker thus (exclusively) updates the lock status to acquire the lock and appends the key to the local list `WIP`.

Lines 11-13: If the key is *not* found in the hash table, this implies that the data is neither locked nor migrated. The worker attempts to acquire a latch on the hash table[5] and insert the key with a value of "in progress" (locked but not migrated) into the hash table. If after acquiring the latch it finds that the key had already been inserted by another worker in between the initial check and the point where the latch was acquired, it releases the latch and runs the same code that would have been run if it had initially found the key in the hash table (line 12). Otherwise, the key is inserted into the local list `WIP` and the migration of that tuple can proceed (line 13).

Algorithm 1 performs the migration of the group, and updates its key in the hash table to a status of `migrated` at line 9.

4.3.5 Migration aborts

When a migration transaction aborts, after the standard database system code is run to handle the abort, BullFrog[5] must inject additional code that tra-

[5]Similar to what we described in Section 4.3.3 for the bitmap, the hash table is partitioned and each partition is protected by a separate latch in order to reduce cross-worker contention that would arise if there were a global latch for the entire hash table. Deadlock does not occur since two latches are never acquired simultaneously.

verses the aborted worker's `WIP` list, and for each element, the corresponding key is accessed in the bitmap or hash table and set to [0 0] in the bitmap or `abort` in the hash table.

Figure 4.2 depicts an example of abort handling for bitmap migrations. Workers w2 and w3 both access the 6th tuple in addition to other tuples. w2 accesses it first and grabs that lock and puts it in its `WIP`, while w3 sees that it is locked and puts it in its `SKIP`. After migrating tuples 4, 6, and 7, but before it commits, w2 gets aborted. This causes the underlying system to undo the insertion of the new tuples in the output tables that were caused by the migration of tuples 4, 6, and 7. At the end of the abort logic, BullFrog iterates through w2's `WIP` and resets the lock and migration statuses of those keys back to [0 0]. When tuple 6's status is reset to [0 0], w3 (which had been looping, waiting for the migration of tuple 6 to complete or abort) can migrate that tuple itself.

BullFrog's status tracking data structures are stored in volatile memory. Upon a crash, they must be reinitialized. While the REDO log is scanned during recovery, for each tuple (or group) that is found in a committed migration transaction, the corresponding status is set to [0 1] in the bitmap or `migrated` in the hashmap.[6]

4.3.6 Joins

As we described in Section 4.3.1, joins can either be 1:1 migrations or 1:n migrations depending on the type of join. For example, a foreign-key/primary-

[6]We have yet to implement this feature in the BullFrog codebase.

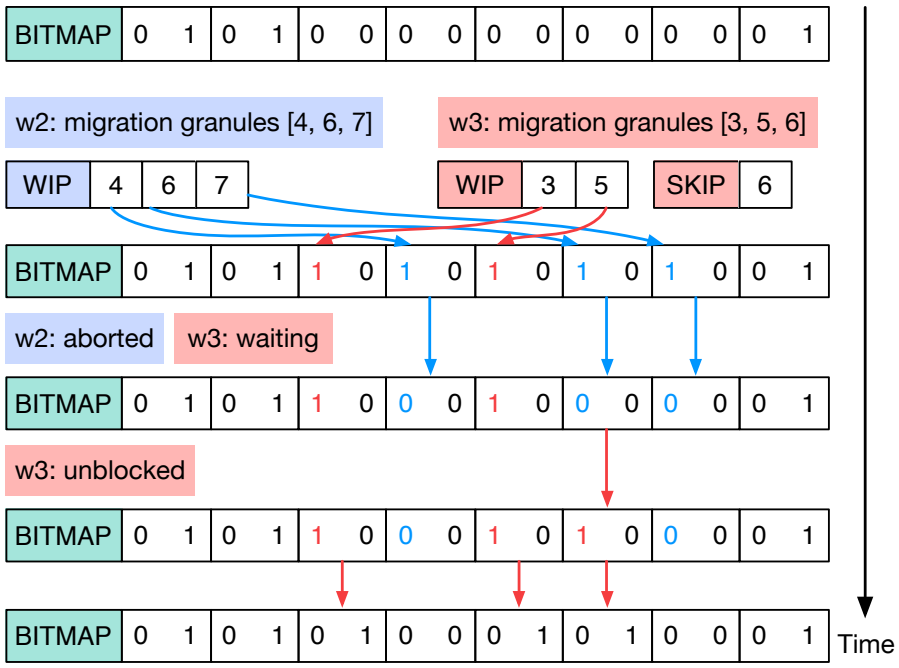


Figure 4.2: Transaction abort handling.

key join is a 1:n migration relative to the primary key input table (PKIT), while at the same time being a 1:1 migration relative to the foreign-key input table (FKIT). However, we said for 1:n migrations, a tuple cannot be considered migrated until the n tuples that it generates have been migrated. Consider a tuple to migrate from the FKIT with a foreign key of 4. Since this is a 1:1 migration relative to the FKIT, the tuple from the PKIT with a primary key of 4 is extracted and joined with this tuple to produce the migrated version of it. After the migrated version has been successfully inserted into the output table, the input tuple in the FKIT can be marked in the bitmap as being migrated. However, the tuple in the PKIT that it joined with (the one with primary key of 4) cannot be considered migrated, since it is a 1:n migration and there may be other tuples in the FKIT with a foreign key of 4. There are two options for what to do next in such a scenario:

(1) *Immediately migrate all other tuples in the FKIT with the same foreign key.* This allows BullFrog to mark the key in the primary key table as `migrated` at the completion of this migration. However, this turns the 1:1 migration on the FKIT side into a n:n migration.

(2) *Stop at this point without adding any additional tuples to this particular migration task.* This option provides BullFrog with more flexibility to migrate lazily, and maintains the simpler 1:1 migration semantics on the FKIT. However, maintaining migration status for the PKIT requires occasional coordination with the FKIT to learn when all tuples with a particular value have been migrated.

In general, the second option is preferable when the cardinality of the foreign key is small or when there is skew such that large chunks of the FKIT would be forced to be migrated at once. In practice, when BullFrog uses the second option in the context of an inner join, it does not attempt to maintain the migration status of the PKIT. Furthermore, it does not maintain the lock status on the PKIT, since the unit of migration is entirely determined by individual tuples in the FKIT, and there are no semantic issues that arise when two different tuples from the FKIT are being migrated concurrently and access the same tuple from the PKIT. If an entire PKIT tuple is to be migrated, then all tuples from the FKIT that it joins with must be locked. Thus, there are no lock status or migration status bitmaps associated with the PKIT. Correspondingly, when using the first option in the context of an inner join, the unit of migration is entirely determined by individual tuples in the PKIT, so BullFrog does not maintain lock or migration bitmaps on the FKIT.

For many-to-many joins, BullFrog provides the same two options for main-

taining the lock and migration status discussed above. However, both input tables are considered a 1:n migration with respect to the other table so it may be impossible to avoid migrating large chunks of data within individual migration tasks if there is skew in the attribute(s) involved in the join condition. Therefore, BullFrog also provides a third option of tracking status based on the combination of tuples from the two tables involved in the join, which increases the granularity of the lazy migration. I.e., instead of x.tupleID $\rightarrow$ (lock_status, migrate_status) it is (x.tupleID, y.tupleID) $\rightarrow$ (lock_status, migrate_status). BullFrog uses the hashmap technique described in Section 4.3.4 to track migration status.

4.3.7 Conflict detection

In some cases, instead of using BullFrog's lock data structures, it would be possible to leverage the underlying database system to prevent duplicate migrations via SQL clauses such as `ON CONFLICT DO NOTHING`. Unfortunately, this method of preventing duplicate migrations has limited applicability. First, the output tables must have a uniqueness constraint declared on an attribute. Many database systems, such as PostgreSQL, require a B-tree index on any attribute declared to be unique. This uniqueness constraint must have been declared on a deterministic attribute whose value is based directly on values of data in the input table(s). Therefore, a primary key generated by an auto-increment function would not be eligible. Even though this primary key is unique, the duplicate insertion during the migration process will not be detected – instead the record will

be inserted twice, with the system generating different unique primary keys for each record.

When applicable, this technique prevents the additional accesses to the old schema on behalf of migration transactions that are blocked, waiting for the old schema to be released. However, it detects conflict at a later stage (at the point of insert into the new schema) and therefore may incur additional wasted work upon a conflict. BullFrog supports both methods for handling migrations and we experimentally compare them in Section 4.4.

4.4 Experimental Evaluation

We implemented a complete prototype of BullFrog on top of PostgreSQL 11.0. Our implementation leverages PostgreSQL's existing view expansion and query rewriting/optimization functionalities – we did not have to modify any core PostgreSQL code. Our bitmap data structures (see Section 4.3.3) use PostgreSQL's existing TIDs for mapping tuples to bits in the bitmap.

The primary goal of our experimental evaluation is to understand the downtime implications of *single-step* migration algorithms in which the database switches from the old to new schema immediately; such single-step migrations historically have required extensive downtime. To this end, we experimentally evaluate the *lazy migration* algorithms of BullFrog and compare their performance under various configurations against *eager migration*. In eager migration, the system immediately physically moves all data stored under the old schema into tables in the

new schema prior to becoming available to client requests over the new schema.

In addition, we also benchmark BullFrog against a multi-step migration implementation in which a schema change is registered with the system ahead of time, and the system copies data into the new schema in a background process. Reads are served from the old schema, while writes go to both schemas. Although BullFrog is targeted for single-step deployment scenarios where giving advanced notification of a schema change is impossible, impractical, or simply too burdensome (see Sections 4.1 and 4.5), there are also some performance differences between single-step and multi-step algorithms that these experiments can illuminate.

For the BullFrog algorithms, we compare solutions that perform duplicate migration detection at time of insert into the new schema via PostgreSQL's `ON CONFLICT` clause (see Section 4.3.7) with solutions that detect duplication prior to generation of the migrated record (see Sections 4.3.3 and 4.3.4).

Workload We developed a variation of TPC-C that includes schema migrations. TPC-C models the transactions involved in placing and delivering orders in a retail application; and querying stock levels of merchandise. The workload is defined by a mix of transactions according to the following percentages: `NewOrder` (45%), `Payment` (43%), `Delivery` (4%), `OrderStatus` (4%) and `StockLevel` (4%). `StockLevel` and `OrderStatus` are external read queries. The schema defined by the TPC-C benchmark consists of nine tables. Our experiments evolve the original schema in various ways, the specifics of which will be discussed in the following

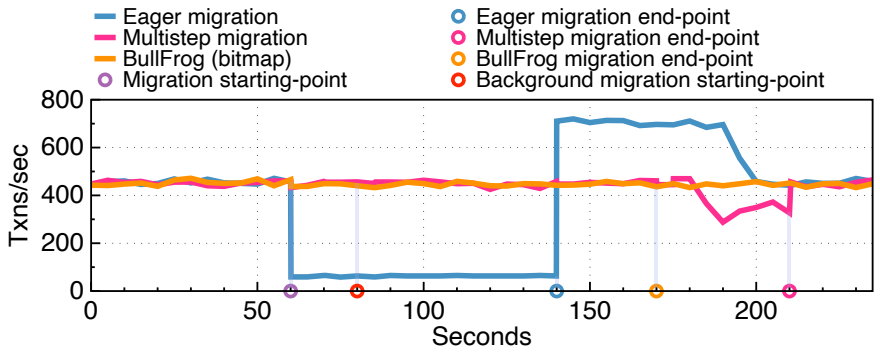
sections.

Experimental Platform We use OLTP-Bench [194] to set up and run our experiments. OLTP-Bench has the ability to support tight control of transaction mixtures, request rates, and access distributions over time. We measure throughput as transactions per second and the end-to-end latency as the time from when the client issues a transaction request until the response is received. Maximum throughput measurements are taken by increasing the rate that clients submit requests until the latency of these requests starts to increase due to queuing delays. The measurements for all of our throughput experiments are averaged over 10 runs, but we found that the variance across runs in each of our experiments was negligible. Latency experiments are presented using cumulative distribution functions (CDFs), and each plot shows a distribution over at least 50,000 points. We run our experiments on an eight-core 2.50GHz Intel Core i7 using 16GB of memory. We dedicate all eight cores to workers within the transaction processing engine.

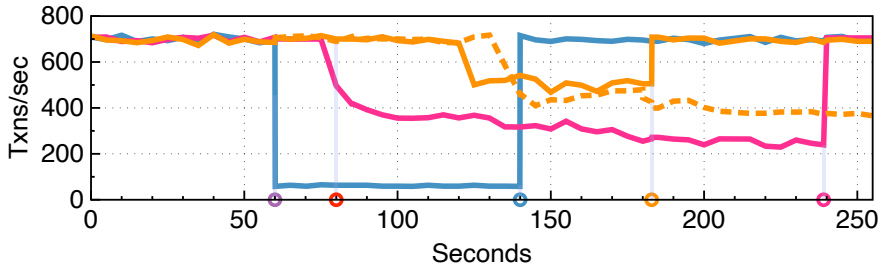
4.4.1 Table split migration

In our first experiment, the baseline TPC-C schema incurs a relatively simple migration: the customer table is split into two tables, where its original set of columns are divided across the two new tables, except for the primary key which appears in both new tables. Each new table contains the same number of tuples as the original customer table. One table includes the customer's personal finan-

cial information (balance, payment, credit, etc.) and the other has the customer's less private information (city, state, zip). Using the terms we described in Section 4.3.1, this is a 1:n migration with respect to the original customer table, since for every row in the customer table there are two rows generated (one row for each of the new tables). Therefore, BullFrog uses a bitmap data structure to track the migration. In this experiment, we use the TPC-C configuration with 50 warehouses, which therefore causes the benchmark to generate 1.5 million records in the customer table. Four out of the five TPC-C transaction types access the customer table—`NewOrder`, `Payment`, `Delivery` and `OrderStatus` and are straightforwardly modified to be compatible with the new customer tables.



Request rate: 450 transactions per second.



Request rate: 700 transactions per second.

Figure 4.3: Throughput during table-split migration.

Figure 4.3 shows how the throughput of transaction processing varies dur-

ing the different phases of the schema migration. Figure 4.3(a) shows the common case where the system is not overloaded at the time of the migration, and the system can devote extra resources to the additional work involved in migrating to a new schema. Figure 4.3(b) shows the same experiment, except that the clients are already submitting requests at the maximum rate at which the system is able to keep up (without falling behind) before the migration, and thus the additional migration work necessarily forces the system to fall behind. The migration begins for all implementations at the purple circle and ends for each system at the later corresponding circles marked in the figure.

Eager migration takes approximately 80 seconds to complete. This time is independent of client request load because all requests that access the customer table during the migration are queued, and the performance of the migration itself is not affected by the size of this queue of waiting transactions. Throughput does not dip all the way to 0 since the `StockLevel` transaction does not access the customer table and can be processed even during an eager migration. When the client load is 450 transactions per second (TPS), there is enough system headroom for the eager migration system to catch up after the migration. This observed in Figure 4.3(a) by a temporary increase in throughput after the migration relative to the throughput before the migration began. When the client load is maxed out at 700 TPS (Figure 4.3(b)), the eager system can never catch up after the migration and cannot decrease the size of the request queue that built up during the migration.

For the lazy migration algorithms, the total time to complete the migration

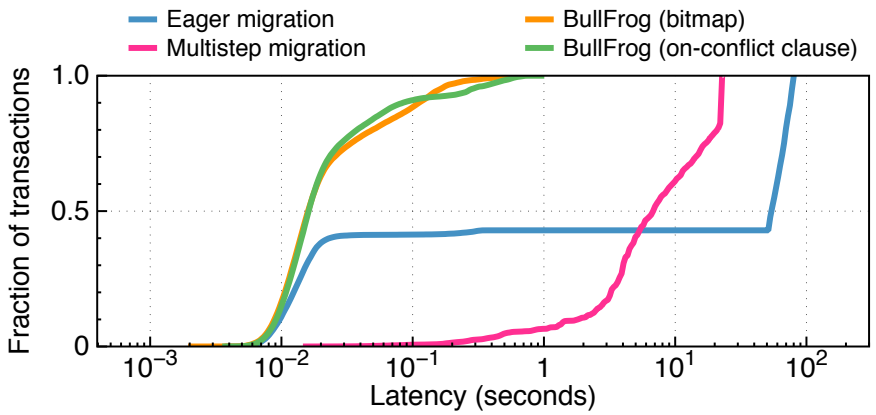
is longer since they process active client requests concurrently with performing the migration. Nonetheless, the background process discussed in Section 4.2.2 enables the migration to complete within the time window shown in the figure. Without the background process (the dotted lines in the figure), the TPC-C benchmark does not access enough distinct tuples to complete the migration within the experimental time window. Throughput steadily degrades as the tables in the new schema become larger and slower to access, while access costs to the fixed-size old schema remains constant despite the increasing percentage of transactions that find out that all relevant tuples have already been migrated.

Lazy migration throughput is unaffected by the migration when the client request rate is at 450 TPS. The additional per-transaction overhead of lazily migrating relevant records to the new schema tables is hidden by the spare capacity of the system. However, when the client load is maxed out at 700 TPS, the throughput is ultimately affected by the migration. At first, the migration overhead is visible in transaction latency, but throughput is not affected. Eventually, OLTP-Bench is forced to queue transactions before sending them to PostgreSQL, and the throughput drops along with the additional queuing latency. The performance of the bitmap vs. on-conflict approaches are similar. In addition to the more steady throughput curves, the lazy migration also performs 13% more transactions overall than the eager migration approach during this experimental window. We attribute this additional efficiency to the improved cache efficiency of bringing in a tuple into cache just once to both migrate it and use it as part of an active client request.

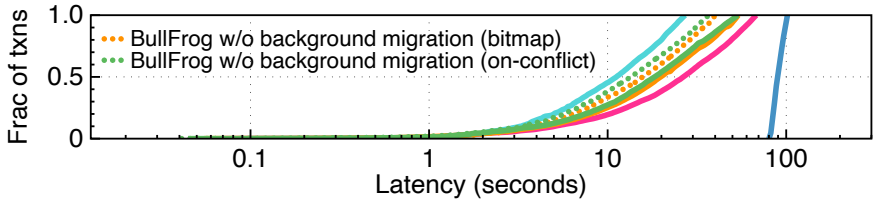
For lazy migration, background migration threads do not begin until 20 seconds after migration initiates, since at first, the client requests themselves are sufficient to keep the migration progress moving along. Only later do the background threads start and search for data to migrate that has not yet been covered by a client request. In contrast, for multi-step migration, the entire migration process happens in the background. Therefore, the background threads start immediately which causes an earlier performance drop.

Surprisingly, in contrast to lazy migration in which the background threads help accelerate the completion of the migration, multi-step migration takes **longer** to complete than lazy migration despite the presence of background threads throughout the migration. The reason is that during the early stages of multi-step migration, most data exists only in the old schema, and updates are only performed in the old schema. However, as migration continues, a larger percentage of data has been migrated to the new schema. Any updates to migrated data must happen twice – in the new and old schema – since the old schema must be able to serve reads until the migration completes. Therefore, as the migration progresses, the multi-step migration needs to perform additional work relative to lazy migration (which never has to perform updates on the old schema). This is observed in the experiments by a steadily dropping throughput for multi-step migration, until the migration completes.

Figure 4.4 shows a CDF of client request latency for the same experiment, starting at the point the migration begins until the end of the experimental window from Figure 4.3. Latency results are plotted for only one transaction type



Request rate: 450 transactions per second.



Request rate: 700 transactions per second.

Figure 4.4: Latency during table split migration.

(the most complex – `NewOrder`) in order to avoid variations due to the different complexities of different TPC-C transaction types. When the client request rate is 700 TPS, the eager migration algorithm is never able to catch up. Thus, the 80 second downtime required to perform the migration is experienced not only by the requests that were submitted during the downtime, but also by the requests that were submitted afterwards, since the size if the request queue never has a chance to decrease. In contrast, at 450 TPS, the eager system is able to catch up. Therefore, the CDF appears as a step – the left side of the graph shows the requests that were submitted after the system catches up, while the right side shows the latency of transactions before the system catches up. The multi-step and lazy schemes, are also never able to catch up when then are no spare resources in the system at 700

TPS. However, because of its superior throughput, BullFrog never gets as far behind as the other algorithms, and both the BullFrog and multi-steps algorithms fall behind at a more steady rate because of their lack of downtime. The poor latency for multi-step at 450 TPS is caused by the throughput dip from Figure 4.3 and resulting queuing delays. Overall, the latency of the lazy algorithms is up to an order of magnitude better than the eager and multi-step algorithms and is comparable to the latency of TPC-C without any migration.

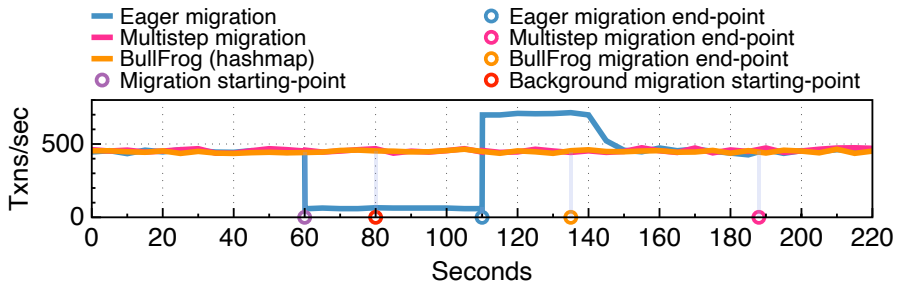
4.4.2 Aggregate Migration

The `Delivery` transaction collects a number of the oldest undelivered orders and marks them as having been delivered. As part of this process, it performs an implicit aggregate operation as follows:

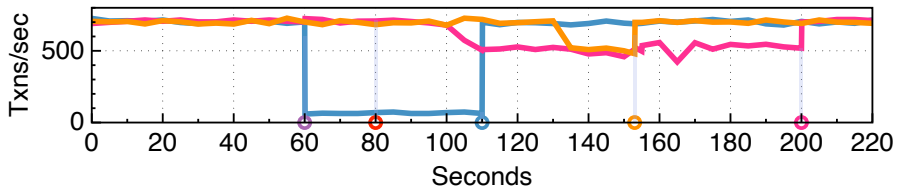
```
SELECT SUM(OL_AMOUNT) AS OL_TOTAL FROM ORDER_LINE
    WHERE OL_O_ID = ? AND OL_D_ID = ? AND OL_W_ID = ?;
```

In our next experiment, we model a schema evolution in which this aggregation is maintained as a separate table. This evolution can be thought of as a materialized view that is maintained by the application instead of the database system. The migration request runs the initial aggregation of the 15 million tuples in the `ORDER_LINE` table, and all future transactions update both the original and aggregated version of this table.

Figures 4.5 and 4.6 show the throughput and latency of this experiment using the same methodology as Figures 4.3 and 4.4 respectively. Using the terminology from Section 4.3.1, this is a n:1 migration with respect to the `ORDER_LINE`



Request rate: 450 transactions per second.



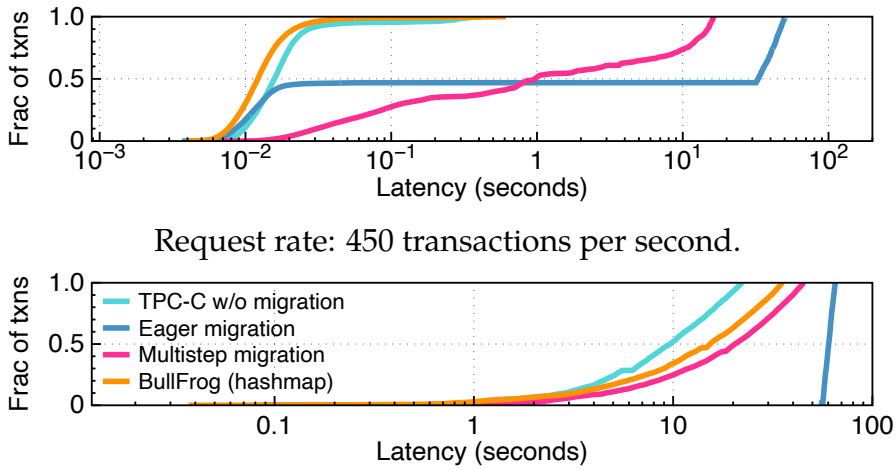
Request rate: 700 transactions per second.

Figure 4.5: Throughput during aggregation migration.

table (in contrast to the previous experiment which was a 1:n migration). Therefore, BullFrog uses a hashmap data structure instead of a bitmap to track the migration. Nonetheless, the results of this experiment are similar to the table-split experiment where throughput and latency are not affected by the migration at 450 TPS, but all systems fall behind at 700 TPS. However, the amount of data that must be written as part of the aggregation migration is smaller (since the output table is small), so the migration is cheaper and the window of throughput reductions for all approaches is smaller and the systems fall less behind.

4.4.3 Join Migration

TPC-C's `StockLevel` transaction is a read-only transaction that scans a warehouse's inventory for items which are (or close to being) out of stock. As part of this process, a join occurs:



Request rate: 450 transactions per second.

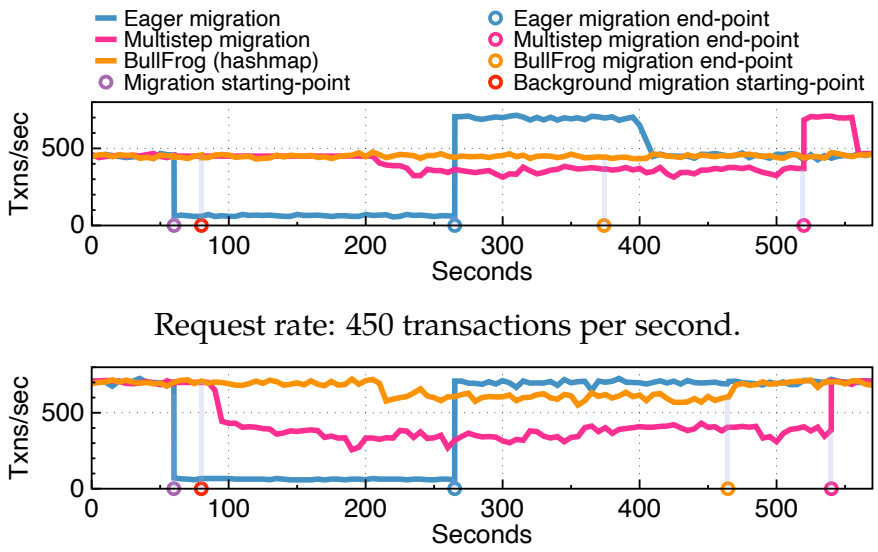
Request rate: 700 transactions per second.

Figure 4.6: Latency during aggregation migration.

```
SELECT COUNT(DISTINCT (S_I_ID)) AS STOCK_COUNT
    FROM ORDER_LINE, STOCK WHERE S_I_ID = OL_I_ID ...;
```

We model a situation where the application developers prioritize the performance of `StockLevel` queries by denormalizing the schema so that the order line and stock tables are already joined. The new schema includes this new table – named `orderline_stock` – instead of the original order line and stock tables. All transactions that accessed the old tables are replaced by new transactions against the `orderline_stock` table that consists of close to 8 million tuples.

Figure 4.7 shows the throughput results of this experiment and Figure 4.8 shows the latency results. This join is the most resource intensive of all the migrations we have experimented with, and thus the throughput dip of all systems, including multi-step migration, is more extended (except BullFrog at 450 TPS which still has no throughput dip). The eager approach experienced over 200 seconds of downtime. The join involved in the migration is a many-to-many join,

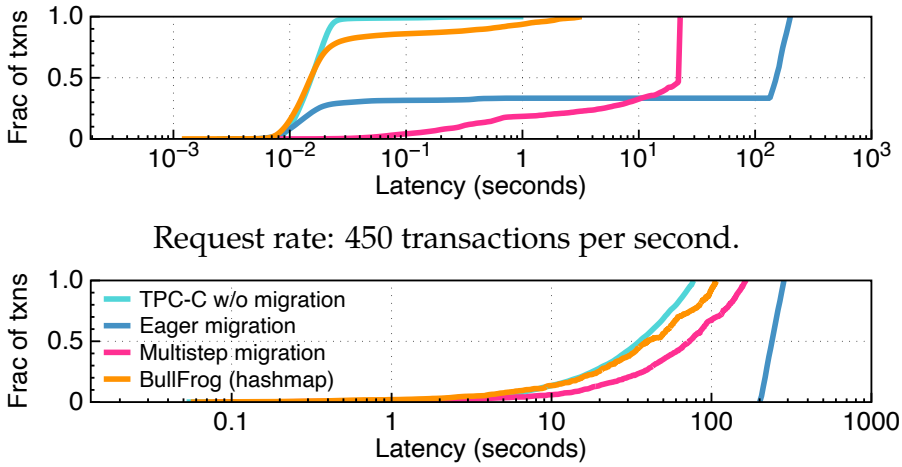


Request rate: 450 transactions per second.

Request rate: 700 transactions per second.

Figure 4.7: Throughput during join migration.

and BullFrog uses the hashmap-based n:n migration approach discussed in Section 4.3.6. When BullFrog attempts to perform the migration during a period of maximum load (700 TPS), latency steadily increases to 10 seconds per transaction, until PostgreSQL reaches its maximum number of concurrent transactions. At this point, throughput dips by approximately 100 TPS as OLTP-Bench queues transactions. After the migration completes, the throughput returns to its original level since the new pre-joined table is designed to accelerate the `StockLevel` transaction which appears relatively infrequently in TPC-C (4% of transactions). Furthermore, the `orderline_stock` table retains all secondary indexes of the two tables that generated it. However, latency never returns to its original level since the system is running at maximum load and can never catch up. The same is true of the eager system, but since it got further behind, the steady latency after the migration is an order of magnitude higher than BullFrog. Thus the lazy approaches



Request rate: 450 transactions per second.

Request rate: 700 transactions per second.

Figure 4.8: Latency during join migration.

are superior to the eager and multi-step approaches, both in terms of the size of the throughput dip and also the increase in latency.

4.4.4 Tracking Overhead

We now investigate some sources of overhead in BullFrog. All experiments in this section use the table split migration.

4.4.4.1 Data Structure Maintenance

We first measure the overhead data structure maintenance in BullFrog by comparing BullFrog performance with a version where no data structures are necessary. Instead, the application is modified such that the `NewOrder` transactions cumulatively access each tuple in the old schema exactly once, rendering migration status tracking unnecessary. Figure 4.9 shows the throughput and latency improvements of removing the tracking data structures is small since they

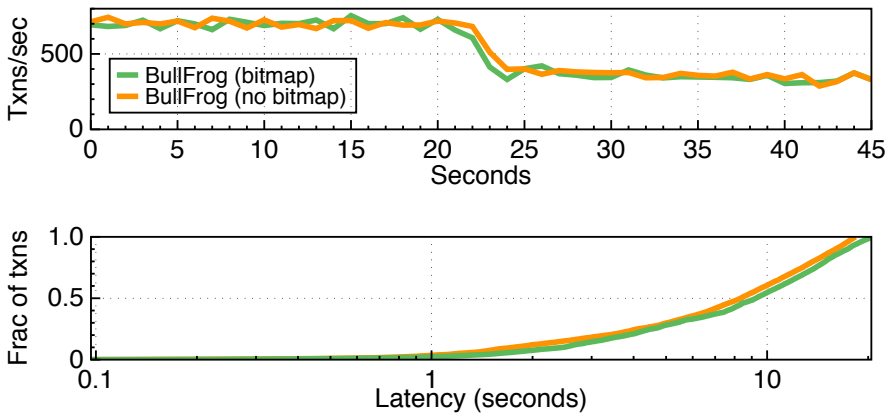


Figure 4.9: Data structure maintenance cost.

do not introduce significant overhead.

4.4.4.2 Lock and Latch Contention

In this experiment, we create a variable number of *hot records* over which transactions exclusively access. Decreasing the size of this hot set increases the contention in the workload, and causes two potential problems for BullFrog. First, it increases the probability of duplicate, simultaneous attempts to migrate a tuple, which causes one of them to block until the first one completes the migration. Second, it increases latch contention for the hot partitions in BullFrog's data structures. Figure 4.10 shows that decreasing the hot set from 1,500,000 to 15,000 records indeed causes a longer drop in throughput during the migration. We verified that this was due to lock contention (and not latch contention) by re-running the same experiment without making transactions wait upon reaching a locked record. We found that the performance drop was due to transactions repeating the loop through the set of records to migrate, waiting for the lock to be

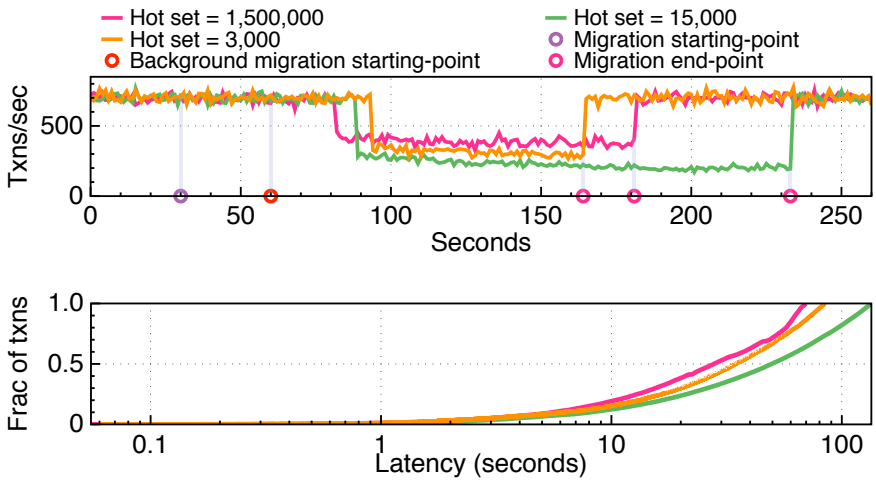


Figure 4.10: Skewed data access.

released (line 10 from Algorithm 1). This increases the latency of the transaction and reduces the number of transactions that can processed concurrently, thereby extending the migration time. However, for very small hot sets, the opposite phenomena is observed. The hot set gets quickly migrated, and the rest of the migration is performed by the background threads which can proceed efficiently and independently with minor impact on throughput.

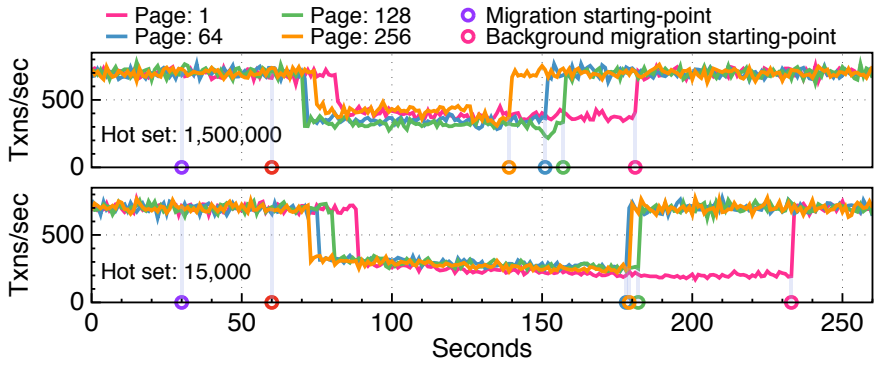
4.4.4.3 Migration Granularity

We next vary the granularity of migration. Instead of tracking migration status at the tuple level, it is done at the page level, and we vary the size of pages and contention in the workload. Figure 4.11 shows that migrating data in larger chunks increases the latency of each operation, but allows the migration to complete more quickly. At 450 TPS, single-tuple granularity is best at low contention, since no matter the granularity, BullFrog can keep up with the request rate, so the

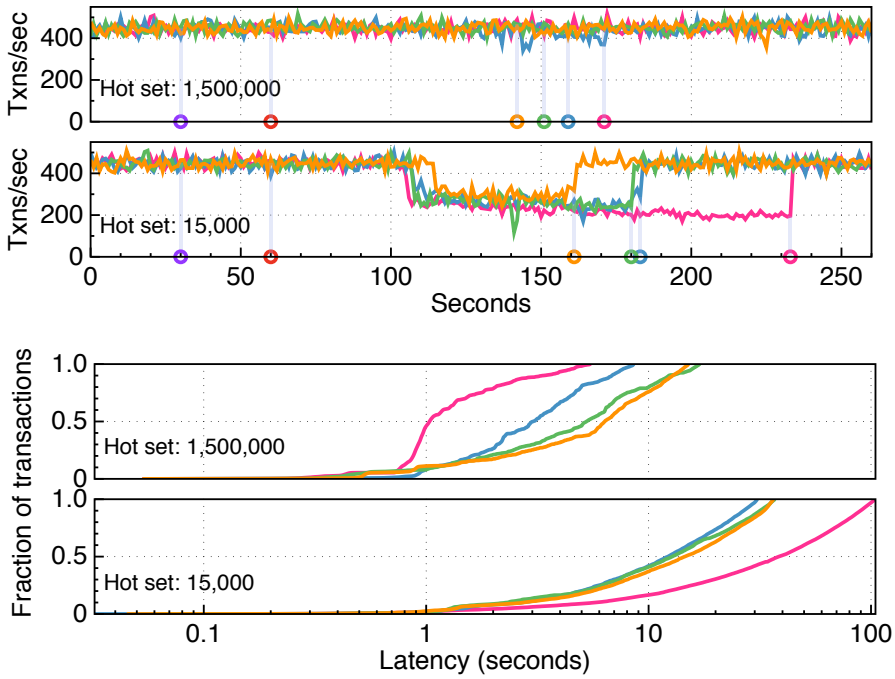
latency advantage of fine granularity is preferable. However, under higher contention, coarse granularity migration is preferred, since the latency benefit of migrating with tuple granularity is negated by the additional queuing delays caused by the extended migration period. This is also the case when running at 700 TPS.

4.4.5 Integrity Constraints

We next evaluate the overhead of constraint preservation during a migration. The TPC-C benchmark includes foreign key constraints from the `Customer` table to `Order` and `District`. In Figure 4.12(a) we remove one (green line) or both (pink line) constraints to observe the improvement in performance from avoiding the overhead of migrating additional data in order to check the constraints. As explained in Sections 4.2.1, 4.2.3, and 4.2.4, the presence of integrity constraints in the new schema limits the laziness in which BullFrog can work, since it must migrate not only the data being accessed by the client request, but also all data necessary to check the integrity constraints in the new schema. Since not all transactions in TPC-C access the customer table, the difference in performance was hard to observe. Therefore, we repeated the same experiment, but removed the transactions that do not access the customer table from the workload. These results are presented in Figure 4.12(b), where the overhead of constraint preservation manifests primarily as an earlier drop in throughput. This is because the additional data that is migrated per transaction limits the number of transactions that can be processed concurrently which accelerates the point at which the DB pushes back



Request rate: 700 transactions per second.



Request rate: 450 transactions per second.

Figure 4.11: Varying access skew and migration granularity. End points are marked by the corresponding circles.

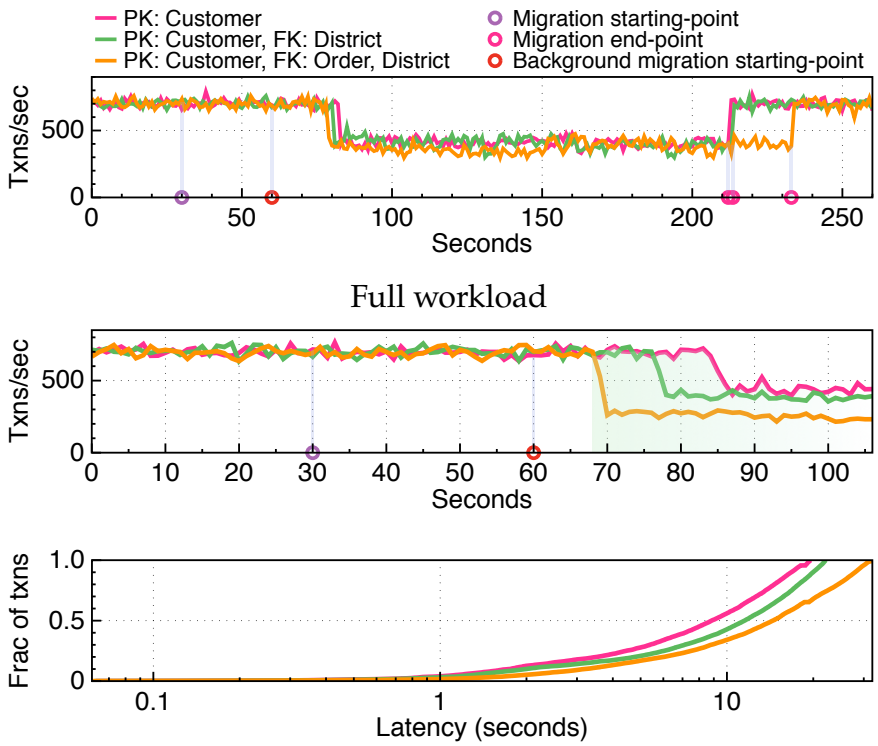


Figure 4.12: `FOREIGN KEY` constraints on table split migration.

on OLTP-Bench to reduce the input workload.

4.5 Related Work

[173] describes online schema evolution using triggers in a telecom database which is not allowed to be down for more than a minute in a year. The paper proposes two different ways to update the schema: soft schema change and hard schema change. For soft schema changes, old transactions are executed using the old schema and new transactions are processed using new schema. Old and new transactions can execute concurrently. For hard schema changes, transactions in the old schema are executed until all of them have finished executing. Thereafter

the system switches over to using the new schema. However, even the hard switch uses a multi-phase process in which triggers are used to prepare the new schema in advance of the switch. In contrast, BullFrog supports single-step schema evolution.

The work on non-blocking schema change in F1 by Google works similarly to the "soft schema change" mechanism described by Ronstrom [184]. Schema changes are done asynchronously across servers. Since F1 is a distributed system with no synchronization across the servers, different servers may transition to the new schema at different times and multiple schema versions may be in use simultaneously. To simplify reasoning about correctness of the implementation, the authors restrict the servers in an F1 instance from using more than two distinct schema versions. Tools such as LessQL [185] facilitate automatic rewriting of queries to use evolved versions of a schema. However, soft schema change solutions restrict the scope of the schema evolution to ensure capability across all active schema versions. In contrast, BullFrog uses a more general approach that does not restrict the scope of the migration operations.

A host of schema migration tools generalize the state-of-the-art multi-step schema migration process, including Percona online schema change [174], Facebook online schema change [175], OAK online alter table [176] and LHM [177]. In the first step, the new schema is registered without yet becoming actively used. Writes performed to a source table (from the old schema) are propagated into a shadow table (for the new schema) that is gradually synchronized in the background using triggers. After all the data has been migrated, the second step in-

volves switching over to the new schema by locking the source table briefly and renaming the shadow table (if necessary) to bring it online. The work on QuantumDB [183] along with the dissertation by [178] use a similar approach which use a combination of materialized views and triggers for maintaining consistency with the original tables while they are updated. In addition to the general disadvantages of multi-step migrations that we discussed in the introduction, all of these tools use update/insert triggers for applying the changes from the old table to the new table. Triggers are known to increase lock contention and at times render the table or the entire database inaccessible due to contention [195].

[179] avoid the use of triggers by using log propagation to perform non-blocking schema transformation. Similarly, Github's online schema change tool `gh-ost` slowly and incrementally copies existing data from the source table to the shadow table while using the binary log stream in MySQL to capture ongoing changes on the source table, and replaying them to the shadow table asynchronously [180]. When the write load gets higher, `gh-ost` can't keep up with binary log, and may not finish at all [196]. Since these techniques read from the log files, there is a delay in between the time the changes are committed in the original table and the time they are applied to the shadow table. Similar to the other multi-step migration techniques we discussed above, these techniques allow queries to execute over the new schema only after the shadow table is caught up to the source.

Schema migration shares some complexities with database migration in which a database is moved from a source node to a destination node, and the copy on

the source node is either kept or deleted after the migration [197, 198, 199, 200]. There also exists lazy implementations of database migration, in which data is pulled to the destination node as it is needed [201, 202], with background processes that ensure all data is eventually migrated, similar in theme to BullFrog. However, these lazy approaches do not make significant changes to the schema during migration. At most, simple 1:1 schema migrations are allowed such as type changes of an attribute. In contrast, BullFrog implements lazy schema migration that supports an arbitrary number of complex schema changes, including 1:n, n:1, and n:n migrations.

Our goal of single-step schema evolution is driven by the software maintenance community. For example, the work on KVolve starts with the same single-step migration requirement [188] and uses a lazy migration approach in the context of migrating an application on top of a NoSQL database (Redis). NoSQL databases are widely used by continuous deployment practitioners since they typically do not enforce schema constraints. Our work on BullFrog proves that lazy migration can be used for traditional relational database systems that enforce schema constraints and require physical data reorganization during a schema migration.

BullFrog's lazy approach to schema migration can be thought of as combining previous work on lazy transaction processing in database systems [203] with transaction decomposition [204, 205, 206] such that a large migration transaction is decomposed into separate smaller transactions that are processed lazily.

4.6 Summary

As applications and database systems increasingly evolve in lockstep, the database system must support single-step migration where the database must instantaneously switch over to a new schema with no downtime. BullFrog succeeds in using a lazy migration approach so that the new schema can be instantaneously ready for access even when the physical data has not yet been migrated to the new schema. Experiments show that BullFrog's lazy approach only causes a slight reduction in throughput and increase in latency during the migration, in contrast to eager approaches that cannot process any transactions during the migration.

Chapter 5: Conclusion and Future Work

5.1 Conclusion

With the growing scale of data and the demands on data analytics, the structured metadata and unstructured byte contents of the files managed within distributed file systems or object stores scale commensurately. We built FileScale, a three-tiered distributed architecture that incorporates a distributed main-memory database system at the lowest layer, along with distributed caching and routing functionality above it, so that most requests can be served with asynchronous, batched interactions with the database layer. FileScale's architecture enables elastic scaling of each layer in the architecture independently.

Stream-processing workloads and modern shared cluster environments exhibit high variability and unpredictability. An ideal event-stream processing service with pay-for-use and SLAs must be performance competitive, scalable, highly available, and low cost. FaaS is a cloud computing service that allows developers to build, compute, run, and manage application packages as functions without maintaining their infrastructure. FaaS is becoming increasingly popular due to its ease of programming, fast elasticity, and fine-grained billing. We built Flock, a cloud-native streaming query engine that enables SQL on FaaS platforms with

heterogeneous hardware with the ability to shuffle and aggregate data without the need for a centralized coordinator or remote storage like S3.

Continuous deployment is the increasingly popular practice of frequent, automated deployment of software changes, with some practitioners deploying multiple changes per day. To realize the benefits of continuous deployment, it must be straightforward to deploy updates to both front-end code and the database, even when the database's schema has changed. Furthermore, as applications and database systems increasingly evolve in lock-step, the database system must support single-step migration where the database must instantaneously switch over to a new schema with no downtime. As a result, we built BullFrog, an extension to PostgreSQL, to support single-step, non-backwards compatible schema migrations without downtime.

5.2 Future Work

According to the Seattle report on database research [207], database systems offered as cloud services are widely used and have witnessed explosive growth. As a result, I believe there will be further tremendous opportunities to build the next-generation cloud data architecture.

Cloud Functions. Cloud customers choose serverless computing because it allows them to stay focused on solving problems unique to their domain or business rather than on server administration or distributed systems problems. In practice, customers realize substantial cost savings when porting applications to

serverless. In addition, the advent of modern FaaS platforms like AWS Lambda [7] and Google Cloud Functions [8] heralded a new way of thinking about cloud-based applications: a move away from monolithic, slow-moving applications toward more distributed, event-based, serverless applications based on lightweight, single-purpose functions where managing underlying infrastructure was a thing of the past.

However, some cloud customers have raised concerns about vendor lock-in, fearing reduced bargaining power when negotiating prices with cloud providers. The resulting switching costs benefit the largest and most established cloud providers and incentivize them to promote complex proprietary APIs that are resistant to de facto standardization. Standardized and straightforward abstractions, such as SQL introduced by Flock, would remove serverless adoption's most prominent remaining economic hurdle.

Differentiation of cloud functions amongst cloud suppliers is becoming more common. Many traditional applications, for example, perform poorly when constrained to a single-request model in FaaS platforms. Google Cloud Functions allows for up to 1,000 concurrent requests on a single instance of an application, providing a far greater level of efficiency [136]. This per-instance concurrency is a game-changer in the distributed query engine for cloud function services, since it enables efficient shuffling. We can extend Flock to popular FaaS platforms and assess which platform is best for developing a high-performance, low-cost query engine.

Data Mesh and Governance. Data Mesh [208] proposes a peer-to-peer ap-

proach to scale out data sharing and consumption aligned with the axes of organizational growth, enable agility by removing centralized bottlenecks of data teams and warehouse or lake architecture, and increase resiliency of the analytics and ML solutions by removing complex data pipelines. Data Mesh is a new approach in sourcing, managing, and accessing data for analytical use cases at scale, which shifts data governance from a top-down centralized operational model with human interventions, to a federated model with computational policies embedded in the nodes on the mesh. There are many unanswered questions here. For example, how can data mesh governance be automated for a consistent, connected, and trustworthy experience?

We hope that continued experience with these systems will help us address the challenges in cloud computing and lead to solutions that are applicable to real-world scenarios.

Bibliography

[1] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016.

[2] Snowflake. `https://www.snowflake.com`.

[3] Google bigquery. `https://cloud.google.com/bigquery`.

[4] Amazon redshift. `https://aws.amazon.com/redshift`.

[5] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*, pages 239–250, 2015.

[6] Netflix Blog. Stream processing with mantis (b. schmaus, et al., 2016), 2017.

[7] AWS Lambda. `https://aws.amazon.com/lambda/`.

[8] Google cloud functions. `https://cloud.google.com/functions`.

[9] Azure functions. `https://azure.microsoft.com/en-us/services/functions/`.

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

[11] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[12] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[13] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, 2019.

[14] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 115–130, 2020.

[15] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 131–141, 2020.

[16] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus:{NIMBLE} task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 653–669, 2021.

[17] Amazon S3. `https://aws.amazon.com/s3/`.

[18] Dong Qiu, Bixin Li, and Zhendong Su. An empirical analysis of the co-evolution of schema and code in database applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 125–135, 2013.

[19] Souvik Bhattacherjee, Gang Liao, Michael Hicks, and Daniel J Abadi. Bullfrog: Online schema evolution via lazy evaluation. In *Proceedings of the 2021 International Conference on Management of Data*, pages 194–206, 2021.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[21] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[22] P. Schwan. Lustre: Building a file system for 1000-node clusters. *In Proceedings of the 2003 Linux Symposium*, 2003.

[23] Michael Abd-El-Malek, William V. Courtright, II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: Versatile cluster-based storage. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[24] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002.

[25] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Martí, and Eugenio Cesario. The XtreemFS architecture - a case for object-based file systems in Grids. *Concurrency and Computation - Practice and Experience*, 2008.

[26] Konstantin V Shvachko. Hdfs scalability: The limits to growth. *; login:: the magazine of USENIX & SAGE*, 35(2):6–16, 2010.

[27] Removing name-node's memory limitation. `https://issues.apache.org/jira/browse/HDFS-5389`.

[28] Haoyuan Li. Alluxio: A virtual distributed file system. `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-29.pdf`, 2018.

[29] Andrew Audibert. Scalable metadata service in alluxio: Storing billions of files. `https://www.alluxio.io/blog/scalable-metadata-service-in-alluxio-storing-billions`, 2019.

[30] Jan Stender, Björn Kolbeck, Mikael Högqvist, and Felix Hupfeld. BabuDB: Fast and Efficient File System Metadata Storage. *2010 International Workshop on Storage Network Architecture and Parallel I/Os*, pages 51–58, 2010.

[31] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *2013 USENIX Annual Technical Conference* (*USENIX ATC 13*), pages 145–156, San Jose, CA, June 2013. USENIX Association.

[32] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248. IEEE, 2014.

[33] Lin Xiao, Kai Ren, Qing Zheng, and Garth A. Gibson. Shardfs vs. indexfs: Replication vs. caching strategies for distributed metadata management in cloud storage systems. In *Proceedings of the Sixth ACM Symposium on Cloud*

Computing, SoCC '15, page 236–249, New York, NY, USA, 2015. Association for Computing Machinery.

[34] Qing Zheng, Charles D Cranor, Danhao Guo, Gregory R Ganger, George Amvrosiadis, Garth A Gibson, Bradley W Settlemyer, Gary Grider, and Fan Guo. Scaling embedded in-situ indexing with deltafs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 30–44. IEEE, 2018.

[35] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 353–369, 2019.

[36] Hdfs federation. `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html`.

[37] Hdfs scalability with multiple namenodes. `https://issues.apache.org/jira/browse/HDFS-1052`.

[38] Hdfs viewfs guide. `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ViewFs.html`.

[39] Pulkit A. Misra, Íñigo Goiri, Jason Kace, and Ricardo Bianchini. Scaling distributed file systems in resource-harvesting datacenters. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 799–811, Berkeley, CA, USA, 2017. USENIX Association.

[40] Hdfs router-based federation. `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs-rbf/HDFSRouterFederation.html`.

[41] Hdfs router-based federation. `https://issues.apache.org/jira/browse/HDFS-10467`.

[42] Bytedance nnproxy. `https://github.com/bytedance/nnproxy`.

[43] Alexander Thomson and Daniel J. Abadi. Calvinfs: Consistent wan replication and scalable metadata management for distributed file systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 1–14, Berkeley, CA, USA, 2015. USENIX Association.

[44] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. Hopsfs: Scaling hierarchical file system metadata using newsql databases. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 89–103, Berkeley, CA, USA, 2017. USENIX Association.

[45] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. Azure data lake store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 51–63, 2017.

[46] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into google's scalable storage system. `https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system`, 2021.

[47] Konstantin V Shvachko and Yuxiang Chen. Scaling namespace operations with giraffa file system. *USENIX; login*, 2017.

[48] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[49] Apache hbase. `https://hbase.apache.org`.

[50] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[51] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[52] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[53] Voltdb. `https://www.voltdb.com`, 2010.

[54] Apache ignite. `https://ignite.apache.org`.

[55] Alternate hash table for namenode memory optimization. https://issues.apache.org/jira/browse/HDFS-1114.

[56] Hdfs high availability using the quorum journal manager. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html.

[57] Hdfs high availability using nfs. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html.

[58] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[59] Nnthroughputbenchmark. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/Benchmarking.html.

[60] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, August 2008.

[61] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael 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, pages 981–992, New York, NY, USA, 2008. ACM.

[62] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615, 2014.

[63] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The Design and Implementation of the Database File System. 2002.

[64] Haryadi S Gunawi, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 131–146, 2008.

[65] Leveldb. https://github.com/google/leveldb.

[66] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system.

In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[67] Rocksdb. `https://github.com/facebook/rocksdb`.

[68] Winfs: Windows future storage. `https://en.wikipedia.org/wiki/WinFS`.

[69] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST'21)*, pages 217–231, 2021.

[70] Muthu Annamalai. Zippydb: A distributed key value store. `https://www.youtube.com/embed/ZRP7z0HnClc`.

[71] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[72] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[73] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

[74] Marshall Kirk McKusick and Sean Quinlan. Gfs: Evolution on fast-forward. *Queue*, 7(7):10:10–10:20, August 2009.

[75] Pavan Edara and Mosha Pasumansky. Big metadata: When metadata is big data. *Proc. VLDB Endow.*, 14(12):3083 – 3095, 2021.

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

[77] Swapnil V Patil, Garth A Gibson, Sam Lang, and Milo Polte. Giga+ scalable directories for shared file systems. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pages 26–29, 2007.

[78] Scaling uber's apache hadoop distributed file system for growth. `https://eng.uber.com/scaling-hdfs/`.

[79] Hdfs router-based federation rebalancer. https://issues.apache.org/jira/browse/HDFS-13123.

[80] Hfr: Rename across federation namespaces. https://issues.apache.org/jira/browse/HDFS-15087.

[81] Michael Cafarella, David DeWitt, Vijay Gadepally, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, and Matei Zaharia. Dbos: A proposal for a data-centric operating system. *arXiv preprint arXiv:2007.11112*, 2020.

[82] Google dataflow. https://cloud.google.com/dataflow.

[83] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.

[84] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[85] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. Flumejava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010.

[86] Amazon kinesis data analytics. https://aws.amazon.com/kinesis/data-analytics/.

[87] New for AWS Lambda – 1ms Billing Granularity Adds Cost Savings. https://aws.amazon.com/blogs/aws/new-for-aws-lambda-1ms-billing-granularity-adds-cost-savings/.

[88] Aws lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.

[89] Google cloud functions quota. https://cloud.google.com/functions/quotas.

[90] Azure functions limits. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook-trigger?tabs=csharp#limits.

[91] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. Nexmark—a benchmark for queries over data streams draft. Technical report, Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.

[92] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. pages 1789–1792, 2016.

[93] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

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

[95] Official rust implementation of apache arrow. `https://github.com/apache/arrow-rs`, 2021.

[96] Apache arrow datafusion query engine. `https://github.com/apache/arrow-datafusion`.

[97] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923, 2015.

[98] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[99] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, et al. Dremel: a decade of interactive sql analysis at web scale. *Proceedings of the VLDB Endowment*, 13(12):3461–3472, 2020.

[100] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, 2021.

[101] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Zongchang (Jim) Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover,

Bo Huang, Yanlai Huang, Zhi (Adam) Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Yalan (Maya) Meng, Prashant Mishra, Jay Patel, Rajesh S. R., Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Ye (Justin) Tang, Junichi Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Gensheng Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divyakant Agrawal, Jeff Naughton, Sujata Kosalge, and Hakan Hacıgümüş. Napa: Powering scalable data warehousing with robust query performance at google. *Proc. VLDB Endow.*, 14(12):2986–2997, jul 2021.

[102] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[103] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 ACM SIGMOD international conference on Management of data*, pages 239–250, 2022.

[104] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813. IEEE, 2019.

[105] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, aug 2013.

[106] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder 0002. Impala - A Modern, Open-Source SQL Engine for Hadoop. *CIDR*, 2015.

[107] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang 0002, Suresh Anthony, Hao Liu 0018, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. *ICDE*, 2010.

[108] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438, 2013.

[109] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, pages 601–613, 2018.

[110] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[111] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, 2003.

[112] Lukasz Golab and M Tamer Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

[113] Amazon s3 update – strong read-after-write consistency. `https://aws.amazon.com/blogs/aws/amazon-s3-update-strong-read-after-write-consistency`.

[114] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[115] Aws lambda: Dead-letter queues for async invocations. `https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html#invocation-dlq`.

[116] Aws lambda asynchronous invocation. `https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html`.

[117] mimalloc. `https://github.com/microsoft/mimalloc`.

[118] snmalloc. `https://github.com/microsoft/snmalloc`.

[119] Flock function dependency tree. `https://bafybeidatseo6ixib6ceujgsuqwaj6hpdrujyfthyrtlhb4g2ibwbz2r3m.ipfs.infura-ipfs.io/`, 2022.

[120] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 symposium on cloud computing*, pages 445–451, 2017.

[121] Pywren. `https://github.com/pywren/pywren`.

[122] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[123] Cranelift Code Generator. `https://github.com/bytecodealliance/wasmtime/tree/main/cranelift`.

[124] Aws lambda api reference: Createfunction. `https://docs.aws.amazon.com/lambda/latest/dg/API_CreateFunction.html`.

[125] Zstandard: a fast lossless compression algorithm. `https://github.com/facebook/zstd`.

[126] Using AWS Lambda environment variables. `https://docs.aws.amazon.com/lambda/latest/dg/configuration-envvars.html`.

[127] Aws lambda api reference: Invoke. `https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html`.

[128] Aws lambda functions powered by aws graviton2 processor. https://aws.amazon.com/blogs/aws/aws-lambda-functions-powered-by-aws-graviton2-processor-run-your-functions-on-arm-and-get-up-to-34-better-price-performance/.

[129] Aws graviton processor. `https://aws.amazon.com/ec2/graviton/`.

[130] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 789–794, 2018.

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

[132] Aws lambda function scaling. `https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html`.

[133] Amazon DynamoDB. `https://aws.amazon.com/dynamodb/`.

[134] Citi bike trip histories. `https://ride.citibikenyc.com/system-data`.

[135] Aws lambda rust runtime.

[136] The next big evolution in serverless computing. https://cloud.google.com/blog/products/serverless/the-next-big-evolution-in-cloud-computing, 2021.

[137] David Yanacek. Avoiding insurmountable queue backlogs. https://d1.awsstatic.com/builderslibrary/pdfs/avoiding-insurmountable-queue-backlogs.pdf?did=ba_card-body&trk=ba_card-body, 2019.

[138] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Nsdi*, volume 10, page 20, 2010.

[139] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84, 2013.

[140] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {usenix} symposium on networked systems design and implementation ({NSDI} 20)*, pages 419–434, 2020.

[141] Aws lambda execution environments. https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html.

[142] Aws lambda api reference: Putfunctionconcurrency. https://docs.aws.amazon.com/lambda/latest/dg/API_PutFunctionConcurrency.html.

[143] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

[144] Adaptive query execution: Speeding up spark sql at runtime. https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html, 2020.

[145] Snappy: a fast compressor/decompressor. https://github.com/google/snappy.

[146] Lz4: a extremely fast compression. https://github.com/lz4/lz4.

[147] Apache beam. https://beam.apache.org.

[148] Nexmark benchmark suite. https://beam.apache.org/documentation/sdks/java/testing/nexmark/.

[149] Aws lambda now supports up to 10 gb of memory and 6 vcpu cores for lambda functions. https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/, 2021.

[150] Aws lambda pricing. https://aws.amazon.com/lambda/pricing/.

[151] Amazon s3 pricing. https://aws.amazon.com/s3/pricing/.

[152] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[153] Managing lambda reserved concurrency. https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html.

[154] Aws step functions. https://aws.amazon.com/step-functions/.

[155] Azure durable functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

[156] Google cloud composer. https://cloud.google.com/composer.

[157] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment*, 2022.

[158] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 328–343, 2020.

[159] Netflix conductor. https://netflix.github.io/conductor/.

[160] Zeebe: A workflow engine for microservices orchestration. https://zeebe.io/.

[161] Fission: Open source, kubernetes-native serverless framework. https://fission.io/.

[162] Fn flow. https://github.com/fnproject/flow/.

[163] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proceedings of the VLDB Endowment*, 13:2438–2452, 2020.

[164] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. {SONIC}: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301, 2021.

[165] Philip A Bernstein, Todd Porter, Rahul Potharaju, Alejandro Z Tomsic, Shivaram Venkataraman, and Wentao Wu. Serverless event-stream processing over virtual actors. In *CIDR*, 2019.

[166] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR-2014–41*, 2014.

[167] Flink stateful functions. `https://nightlies.apache.org/flink/flink-statefun-docs-release-3.2/docs/concepts/distributed_architecture/`.

[168] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[169] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software technology*, 57:21–31, 2015.

[170] Mojtaba Shahin, Muhammad Ali Babar, Mansooreh Zahedi, and Liming Zhu. Beyond continuous delivery: an empirical investigation of continuous deployment challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 111–120. IEEE, 2017.

[171] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at facebook and oanda. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE, 2016.

[172] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1327–1342, 2015.

[173] Mikael Ronstrom. On-line schema update for a telecom database. In *Proceedings of 16th International Conference on Data Engineering (ICDE)*, pages 329–338. IEEE, 2000.

[174] Percona Online Schema Change. `https://www.percona.com/doc/percona-toolkit/2.2/pt-online-schema-change.html`, 2016.

[175] Facebook Online Schema Change. https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932/, 2010.

[176] OAK Online Alter Table. https://shlomi-noach.github.io/openarkkit/oak-online-alter-table.html, 2010.

[177] Large Hadron Migrator. https://github.com/soundcloud/lhm, 2012.

[178] Yu Zhu. *Towards Automated Online Schema Evolution*. PhD thesis, UC Berkeley, 2017.

[179] Jørgen Løland and Svein-Olaf Hvasshovd. Online, non-blocking relational schema changes. In *International Conference on Extending Database Technology (EDBT)*, pages 405–422. Springer, 2006.

[180] GitHub Online Schema Change. https://github.com/github/gh-ost, 2016.

[181] Carlo A Curino, Hyun J Moon, MyungWon Ham, and Carlo Zaniolo. The prism workwench: Database schema evolution without tears. In *IEEE 25th International Conference on Data Engineering (ICDE)*, pages 1523–1526. IEEE, 2009.

[182] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *The VLDB Journal*, 22(1):73–98, 2013.

[183] Michael de Jong, Arie van Deursen, and Anthony Cleve. Zero-downtime sql database schema evolution for continuous deployment. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 143–152. IEEE, 2017.

[184] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. Online, asynchronous schema change in f1. *Proceedings of the VLDB Endowment*, 6(11):1045–1056, 2013.

[185] Ariel Afonso, Altigran da Silva, Tayana Conte, Paulo Martins, João Cavalcanti, and Alessandro Garcia. Lessql: Dealing with database schema changes in continuous deployment. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–148. IEEE, 2020.

[186] Michael Stonebraker, Dong Deng, and Michael L Brodie. Database decay and how to avoid it. In *IEEE International Conference on Big Data (Big Data)*, pages 7–16. IEEE, 2016.

[187] Eric A Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[188] Karla Saur, Tudor Dumitraş, and Michael Hicks. Evolving nosql databases without downtime. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 166–176. IEEE, 2016.

[189] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 573–585, 2019.

[190] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for java on a stock JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, volume 49, pages 103–119, 2014.

[191] Christopher M Hayden, Karla Saur, Edward K Smith, Michael Hicks, and Jeffrey S Foster. Kitsune: Efficient, general-purpose dynamic software updating for c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):1–38, 2014.

[192] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[193] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[194] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[195] Why Triggerless? `https://github.com/github/gh-ost/blob/master/doc/why-triggerless.md`, 2016.

[196] Gh-ost benchmark against pt-online-schema-change performance. `https://www.percona.com/blog/2017/07/12/gh-ost-benchmark-against-pt-online-schema-change-performance/`, 2017.

[197] Database migration: Concepts and principles (Part 2). `https://cloud.google.com/solutions/database-migration-concepts-principles-part-2`, 2020.

[198] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment*, 4(8):494–505, 2011.

[199] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. "cut me some slack" latency-aware live migration for databases. In *Proceedings of the 15th international conference on extending database technology (EDBT)*, pages 432–443, 2012.

[200] Takeshi Mishima and Yasuhiro Fujiwara. Madeus: database live migration middleware under heavy workloads for cloud environment. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 315–329, 2015.

[201] Aaron J Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 301–312, 2011.

[202] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. Prorea: live database migration for multi-tenant rdbms with snapshot isolation. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*, pages 53–64, 2013.

[203] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 15–26, 2014.

[204] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, 1995.

[205] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 276–291, 2013.

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

[207] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, et al. The seattle report on database research. *ACM SIGMOD Record*, 48(4):44–53, 2020.

[208] Z. Dehghani. *Data Mesh: Delivering Data-Driven Value at Scale*. O'Reilly Media, Incorporated, 2022.