IMPROVING CLOUD DATA PROCESSING AND STORAGE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ziheng Wang

August 2025

Abstract

SQL is not merely a query language – it is a state of mind. To think in SQL is to view reality through the lens of sets and predicates. A crowded room becomes a table of persons, each with attributes that can be filtered, grouped, and aggregated. Conversations become transactions, friendships become foreign keys, and communities emerge from inner and outer joins. We normalize our thoughts, decomposing complex ideas into atoms that can be recomposed through relational algebra. We seek primary keys in every domain – those unique identifiers that anchor understanding. We think in terms of constraints and integrity, recognizing that truth emerges not from individual records but from the relationships between them.

Each computing epoch has demanded its own translation of this relational philosophy into silicon and wire. From mainframes executing batch jobs to client-server architectures, each generation has reimagined how to manifest set-theoretic operations in the medium of their time. Today, cloud computing presents us with new primitives: ephemeral compute, disaggregated storage, and elastic scale. Our challenge is not to abandon or even evolve the relational creed, but to discover how its eternal truths can flourish when tables grow to petabytes, when compute materializes on demand, and when the "database server" dissolves into a constellation of different hosted services.

This dissertation explores how to realize the relational vision in the cloud era. We begin by improving distributed query processing through two key innovations: balancing fault recovery with pipelined execution in streaming dataflow systems, and reasoning about query execution on heterogeneous compute resources. We then turn to the storage layer, showing how to optimize cloud-native data lakes for selective queries by building consistent, bolt-on indices over object storage. We demonstrate these principles through a concrete implementation for log search, showcasing how relational operations can efficiently navigate massive volumes of semi-structured data.

We hope the reader will come to appreciate how the synthesis of distributed systems theory and cloud engineering practice allows the relational model to flourish beyond its traditional confines without sacrificing its essential beauty.

Acknowledgments

Everyone, especially my advisor Alex Aiken, for their extraordinary patience.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and Background

Two important categories of data processing are OLTP (online transactional processing) and OLAP (online analytical processing). The former concerns itself with handling real-time operational tasks such as managing customer orders, processing payments, and updating inventory levels. OLTP systems are optimized for quick, atomic transactions and maintaining data consistency across multiple concurrent users. These systems typically deal with many small, discrete transactions that modify only a few records at a time, making them ideal for day-to-day business operations where speed and accuracy are crucial. The latter focuses on analyzing large volumes of historical data to support strategic decision-making and business intelligence. OLAP systems are designed to handle complex queries across vast datasets, often aggregating information from multiple sources to identify trends, patterns, and relationships. Unlike OLTP's rapid individual transactions, OLAP operations typically involve reading large amounts of data and performing sophisticated calculations, such as generating sales forecasts, analyzing customer behavior over time, or creating multidimensional reports for executive dashboards. There is an emerging field of data processing called HTAP (hybrid transactional analytical processing) that aims to build unified systems with both OLTP and OLAP capability.

This thesis mostly concerns itself with OLAP workloads. In the Chapters 2 and 3, we will discuss efforts to improve cloud data processing [143, 144]. In the next two chapters, we will discuss efforts to improve storage [145, 146].

Two main challenges facing OLAP workloads currently are:

- An ever increasing volume of data. While a decade ago papers used 1TB datasets as an example of "big data" [151], modern data analytics can easily process over 100TBs of data.
- New data workloads involving novel modalities such as vector embeddings, images and text.

The database community has mostly tried to tackle these challenges by leveraging new compute primitives arising from the maturation of cloud computing, with tremendous academic and industrial

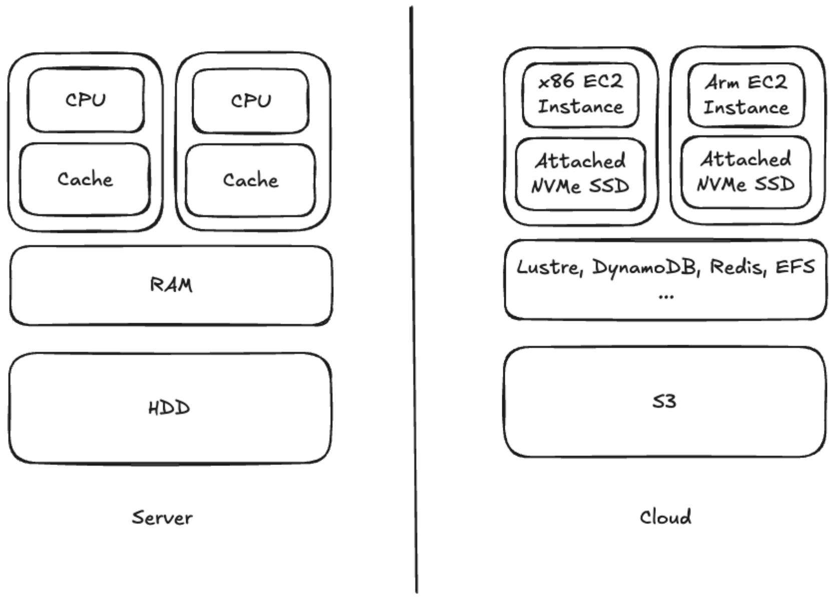


Figure 1.1: Multicore servers (left) vs distributed systems on the cloud (right).

impact. The first cloud native data warehouse, Snowflake, is now a 60B company (as of Jan 2025), while cloud-native data technologies has become a mainstream research topic [48].

1.1 Cloud Primitives

While traditional OLAP data warehouses are designed to work on multicore servers shown on the left of Fig 1.1, cloud OLAP systems are designed to work as distributed systems on cloud platforms such as Amazon Web Services (AWS). Such distributed systems rely on core services these public clouds provide, such as blob storage (AWS S3, Google GCS, Azure Blob Storage) and elastic compute (AWS EC2, GCE, Azure VM).

Multicore servers typically consist of a fixed number of cores with a shared RAM and hard drive. Each core has caches that accelerate access to core-local data; the HDD provides a reliable storage medium; the RAM provides fast ephemeral storage that can be shared between cores.

At first glance, distributed systems on public cloud share a striking semblance to multicore servers. The HDD is replaced by reliable object storage services like AWS S3. Instead of a fixed number of cores managed by an operating system, distributed systems rely on an elastic number of virtual machines (VMs) managed by a container orchestration system like Kubernetes. Distributed caching services like DynamoDB or Redis play a similar role as RAM. Analogous to how caches speed up core-local data accesses in multicore machines, cloud VM instances might offer instance-attached NVMe SSD disks that are much faster to access than object storage or distributed caching services.

1.1.1 Scalable Compute

Cloud distributed systems rely on *elastic* VM instances that can be spun up and down on demand as compute resources. These services (e.g. AWS EC2), empowered by virtualization, containerization and orchestration technology advances in the last decade [34, 40] and gigawatt-scale data centers packed with the latest hardware, allow distributed systems to rapidly scale up and down their compute requirements, such as compute cores or memory, only paying for the resources used.

While a program written for multicore servers executes the same sequence of instructions on different cores of the same machine, distributed systems typically execute the same program over hundreds to thousands of VM instances. Similar to how optimizing software for multicore servers could involve performance engineering single-core code to improve cache efficiency, optimizing distributed systems also involves optimizing the code run on each VM to properly use multiple cores and instance-attached NVMe SSDs. However, there are two major qualitative differences between cores in a multicore server and elastic cloud VMs in a distributed system.

- **Elasticity**: while it is almost impossible to add new cores to servers during program execution, it is trivial (and often expected) to dynamically change the number of VM instances involved in a distributed system execution. While the user can request more resources for the system, the cloud provider or container orchestration system might also decide to preempt resources from the system due to the demands of other customers or other jobs.
- **Heterogeneity**: while the cores in a multicore server are typically identical, distributed systems in public clouds can typically make use of a combination of instance types with different resources in the same job. For example, there could be a mix of VM instances based on x86 and Arm cores in the same job. This situation has become increasingly common, especially at larger organizations, to take advantage of all available compute resources.

While characteristics like preemption place new fault tolerance challenges on cloud OLAP systems, the ability to scale up and cluster heterogeneity also offer new opportunities to make cloud OLAP systems more cost efficient. This thesis explores improvements in fault tolerance mechanisms for distributed OLAP systems and leveraging heterogeneous clusters with a research OLAP system called Quokka in Chapters 2 and 3[1].

1.1.2 Reliable Storage

In traditional database research, the disk is taken as a reliable storage medium. For distributed systems in the cloud, this role is taken by an object storage service (AWS S3, Google Cloud Storage, Azure Blob Storage), which typically boasts extremely high reliability. Object storage is more similar to classic HDDs than the SSDs commonly used today:

[1] https://github.com/marsupialtail/quokka

- **Immutable writes**. While SSDs and HDDs typically support some form of random access writes, object stores only support append only writes: only new objects can be created while existing objects cannot be modified. Like SSDs and HDDs, object stores do support random reads.
- **Performance**. While modern disks typically offer sub 10ms random read latency, object storage first byte read latency is typically around tens of milliseconds, with some requests taking up to 100 ms, slower than the typical seek time in HDDs. However, if accessed properly, object storage boasts virtually unlimited read throughput.[2]

Crucially, similar to hard drives, object storage is *extremely* cheap, costing only around $20 to store a terabyte of data for a month.[3] This cost is impressive especially considering the extremely high availability guarantees – in practice, it is rare for an IT organization to complain about object storage cost, as it is typically eclipsed by other line items in its budget. There is also no limit in the amount or duration of data one can store – it is not uncommon for organizations to store multiple petabytes of data for years for compliance reasons. The cost for storing 1PB for a year is roughly $240,000, not even the fully loaded cost of a single software engineer. Storage is especially cheap compared to some (perhaps overpriced) compute services: e.g. AWS Athena would charge $5/TB read for a single query, the same as storing the data for a week!

The high reliability of cloud storage systems greatly simplifies the design of data systems – whereas previously distributed databases had to take great care to maintain data reliability by replicating data, cloud data systems can guarantee reliability by simply storing the data in object storage. Another topic of this thesis is enabling efficient analytics of novel modalities such as vector embeddings and text leveraging object storage in Chapters 4 and 5 with a research system called Rottnest[4].

1.2 Decoupling Compute and Storage

Traditional database systems for multicore servers tightly couple compute and storage resources, where each database node would manage and process data stored on its local disks. Modern cloud OLAP systems have shifted toward distributed systems that typically separate compute and storage: the storage layer consists of cloud object stores that durably maintain all data, while the compute layer comprises stateless query processing nodes that can be independently scaled. This system architecture allows for more flexible resource management and better cost efficiency, as organizations can provision compute resources based on their immediate processing needs without being constrained by their total data volume.

[2]TB/s throughput can be achieved with proper load balancing across availability zones on AWS as of 2025.
[3]AWS S3 Standard Tier, 2025
[4]https://github.com/marsupialtail/rottnest

The separation of compute and storage in modern cloud OLAP systems is a natural architectural choice given the aforementioned distinct characteristics of elastic compute and reliable object storage. While object storage services provide virtually unlimited, highly reliable storage, elastic VM instances offer scalable, pay-as-you-go compute resources that can be dynamically adjusted. This separation allows OLAP systems to independently scale their compute resources based on query workload demands without being constrained by data locality or storage capacity considerations. In Chapter 3, we will show how query engines can exploit this separation with heterogeneous clusters by assigning different query stages to instance types best suited for the computation (e.g., memory-intensive operations to high-memory instances, compute-intensive operations to CPU-optimized instances).

A consequence of this architectural choice is that while historically the query engine and storage format of databases were highly coupled, e.g. it would be unusual for Postgres to be able to read data in MySQL, query engines and storage formats in cloud databases have grown disparate as separate projects, especially in the open source community. Most open source cloud-native query engines today, such as SparkSQL and Presto do not concern themselves with storage, and are expected to support many different storage formats [30,124]. Similarly, open source cloud storage formats, often referred to as data lakes like Apache Iceberg or Delta Lake, are expected to support different query engines. While proprietary systems like Snowflake and BigQuery typically have their own storage formats, they are moving to support open formats [88,98]. In Chapter 4 and 5, we will show how the decoupling of storage and compute allows us to build lazy indexing systems that are "bolt-on" to existing data lakes.

1.3 Processing: Distributed Query Engines

We now present a more detailed summary of the state-of-the-art in distributed query engines. We present two challenges, fault tolerance and cluster heterogeneity, along with an overview of our proposed solutions.

1.3.1 Stagewise and Pipelined Query Engines

Some examples of distributed query engines are Apache Spark, Google's Dremel (later commercialized as BigQuery), and Meta's Presto [30,99,124]. These systems typically support SQL-based analytics at scale by breaking down complex queries into distributed execution plans. When a user submits a SQL query, these engines first generate a logical query plan, which is then transformed into a physical execution plan that can be distributed across many VM instances. For example, when processing a query that joins two large tables and performs aggregation, the engine might split the data into partitions, distribute join operations across multiple executors, and then combine results through a distributed aggregation phase. Each worker node in the cluster processes its assigned portion of the data, typically fetched from cloud object storage like Amazon S3, with the query engine coordinating the flow of data between stages and handling fault tolerance.

While many known implementation techniques in multicore single-node OLAP data warehouses can be readily transferred to this new setting, e.g. how to break up SQL query execution graphs into parallel shards, cloud-based distributed query engines face some new fundamental architectural challenges. While multicore systems attempt to minimize cross-core communication due to NUMA issues, distributed query engines must coordinate data movement across independent VM instances that may be scattered across different machines across a datacenter, making communication much more expensive compared to computation.

The high cost of communication has given rise to programming paradigms like MapReduce, which the first generation distributed OLAP SQL engines were built on [51, 63, 74]. MapReduce addresses the communication bottleneck by structuring computation into distinct Map and Reduce phases, where data is first processed locally on each machine during the Map phase to minimize data volume before a separate shuffle operation moves data between machines during the Reduce phase. This approach minimizes network communication in several ways:

- **Data locality**: Map tasks are preferentially scheduled on machines that already have the input data locally available.
- **Early filtering**: Map operations can reduce data volume before the shuffle phase by filtering and transforming data locally.
- **Bounded communication**: Each piece of data is typically communicated only once during the shuffle phase, rather than making multiple round trips between machines.
- **Batched communication**: Data transfer happens in bulk during the shuffle, rather than through many small network requests. This bulk data transfer can then be optimized as a distributed primitive.

Most importantly, MapReduce encapsulates these advantages into a simple to understand bulk-synchronous-parallel (BSP) programming model that executes one map-reduce stage at a time. First generation OLAP systems like Hive, Pig and Spark are all built on this programming model, though some systems like Spark evolved beyond the original MapReduce system itself with optimizations such as lineage tracking and in-memory computing [51, 63, 74, 152]. While scalable and simple to program, the BSP-based OLAP systems' stagewise execution model leaves performance on the table. When a job is executing the shuffle stage, it is typically network-bound, leaving the CPUs in the cluster idle. The reverse situation happens in compute-intensive map stages.

A second generation of distributed query engines purpose-built for SQL often employ a pipelined architecture where multiple stages can execute concurrently [48, 124]. This is inspired by high performance multicore OLAP systems, which have largely moved to a pipelined work-stealing morsel based parallelism approach [89]. In some of these systems, the dependencies between tasks in different stages are statically determined [124]. In other more recent systems, such dependencies are dynamically determined to take advantage of work stealing and cache efficiencies [31]. Compared to first generation OLAP engines, these new pipelined engines can fully exploit pipeline parallelism between stages, leading to greater performance and resource utilization.

1.3.2 Challenge: Fault Tolerance

However, due to their more complicated execution model, pipelined query engines typically do not support efficient intra-query fault tolerance, required to harness the elastic nature of cloud compute. While most cloud query engines based on object storage are fault tolerant to data loss, we focus here on intra-query fault tolerance: the query engine can reuse intermediate results to recover faster than restarting the entire query after some tasks fail or have been preempted by the cloud provider or the cluster orchestration software. Current pipelined query engines either re-execute failed queries from the beginning [31, 48], or rely on high-overhead approaches such as durably persisting shuffle partitions between stages [7].

In Chapter 2, we propose a very simple fault tolerance strategy for these pipelined OLAP engines called **write-ahead lineage**. Similar to how a write-ahead log in an OLTP database persists proposed data changes on disk before a transaction is committed, write-ahead lineage persists the dynamically generated task dependencies in pipelined query engines in an ACID data store.

During standard execution, the system stores only kilobytes of lineage metadata as overhead. Tasks are restricted to processing intermediate results only after their lineage data has been durably stored. This maintains the performance benefits of a pipelined query engine with minimal overhead, leading Quokka to outperform Spark by around 2x on TPC-H. We show that combined with a distributed pipeline parallel recovery strategy, we can match the fault recovery performance of Spark as well.

1.3.3 Challenge: Cluster Heterogeneity

While public cloud providers offer a broad range of vastly different VM instance types, from memory-optimized instances with terabytes of RAM to compute-optimized instances with high-performance CPUs to storage-optimized instances with fast local NVMe storage, traditional query optimization research has focused primarily on homogeneous clusters where all nodes have identical resources. Different stages of query execution often have vastly different resource requirements – for example, hash joins are typically memory-intensive while aggregations are often CPU-bound, suggesting potential benefits from matching query operations to specialized instance types.

In Chapter 3, we explore how we can improve the cost efficiency of query engines via mixing different instance types within the same query, developing novel optimization techniques that consider both the resource requirements of different query stages and the cost-performance characteristics of various instance types to minimize overall query cost while maintaining performance objectives.

1.4 Storage: Data Lakes on Object Storage

Let's now turn our attention to storage. In the last decade, object storage has become the de facto storage medium for cloud OLAP engines. The field has converged around the "data lake" concept, where data is stored in columnar storage formats, typically Parquet files, which are in turn organized into a "data lake", which might contain additional metadata files.[5]

1.4.1 Parquet and Data Lakes

We will first describe the format of Parquet files, then cover how data lakes organize Parquet files to facilitate efficient data updates and querying. Several popular data lake formats exist at the time of writing, including Delta Lake, Iceberg and Hudi [24, 25, 27], all of which are based on Parquet files.

In contrast to classic database storage formats (e.g. Postgres) which store tabular data row by row, Parquet is a **columnar format**. It stores data column by column, allowing for efficient compression and retrieval of only the necessary data. More details are shown in Figure 1.2.

Parquet is particularly suitable for cloud object storage:

- **Compression**: Each column can be compressed independently using algorithms best suited for its data type: e.g. dictionary encoding for low-cardinality string columns or run-length encoding for columns with repeated values. This type-specific compression, combined with the natural redundancy in columnar storage, typically achieves better compression ratios than row-oriented formats.

- **Efficient I/O**: By storing column values contiguously, Parquet minimizes I/O when queries access only a subset of columns, which is particularly important for object storage where each GET request incurs latency overhead.

- **Statistics and Metadata**: Each column chunk maintains statistics, such as the minimum and maximum values of the colum. This enables query engine optimizations such as predicate pushdown, where entire row groups can be skipped if the statistics incidcate that none of the values contained would pass a filter condition. These statistics are stored in the file footer, which can be read with a single small request.

These properties, combined with its widespread adoption across the analytics ecosystem, have made Parquet a de facto standard for analytical data storage in the cloud. At first, query engines

[5]The term "data lake" has been taken to mean many things, e.g. a repository for PDFs, images, videos, magic fairy dust, etc. In this dissertation, we will take it to mean metadata formats like Apache Iceberg and Delta Lake.

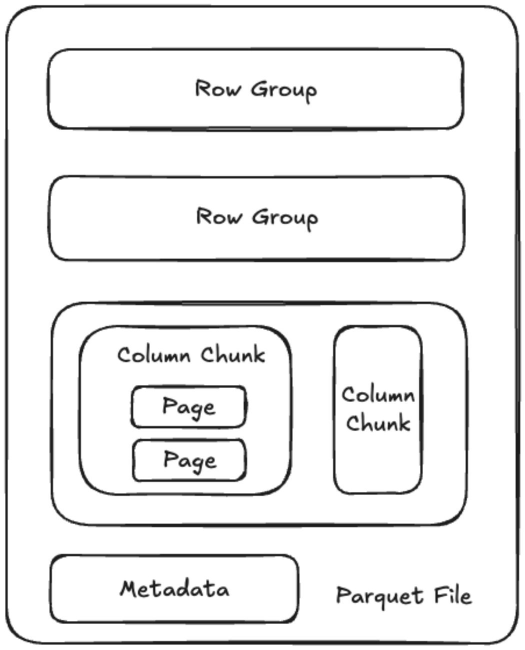


Figure 1.2: The layout of a Parquet file. The format employs a hierarchical structure with multiple levels of organization. At the highest level, a Parquet file consists of row groups, each containing data for a range of rows in the underlying table. Typical Parquet writers target 128MB row group sizes. The row groups contains multiple column chunks. Each column chunk stores the values for a specific column across the rows in that row group, along with metadata like statistics (min/max values, null counts) that enable predicate pushdown. Within column chunks, data is further divided into pages, which are the basic units of compression and encoding. Typical Parquet writers use a page size of 1MB before compression, which usually results in a compressed page size of around 300KB. This hierarchical structure allows for efficient data skipping: queries can leverage metadata at both row group and page levels to skip reading data that won't contribute to query results.

manipulated Parquet files directly stored in directly on object storage in organization structures such as Hive. However, the shortcomings of this approach soon became evident: listing the Parquet files to read became bottlenecks on large queries; write queries overwrote each others' data, etc.

Data lake formats soon emerged that promised strong data consistency through ACID transactions and data versioning on top of the Parquet files. In effect, they offered typical OLTP-like features on top of the raw Parquet files in object storage by keeping additional metadata such as a transactional log stored in Json format.

These data lake formats generally follow similar architectural patterns to provide ACID guarantees and versioning capabilities on top of immutable object storage. At their core, they maintain a transaction log or commit history that tracks changes to the underlying Parquet files over time. Each transaction creates a new "snapshot" of the table, referencing the set of Parquet files that comprise the table's state at that point in time. This append-only design aligns well with object storage's immutable nature - rather than modifying existing files, updates and deletes are handled by creating new files and updating metadata to reference them.

In addition to ACID guarantees, this metadata-driven approach enables several other key features that were difficult or impossible with raw Parquet files:

- **Time Travel**: Since each transaction creates a new snapshot, queries can read from any historical version of the table.
- **Schema Evolution**: New columns can be added while maintaining backward compatibility.
- **Efficient File Management**: Background processes can compact small files to speed up future queries and clean up Parquet files corresponding to obsolete versions while ensuring active queries aren't affected.

Modern data lake query engines have made remarkable advances in query performance on open data lake formats, often matching or exceeding the performance of systems with proprietary storage formats. Through sophisticated optimizations like intelligent file pruning, adaptive query execution, and vectorized processing, these engines can execute complex SQL queries on Parquet files with impressive efficiency. For example, Databricks' Photon engine employs advanced techniques like SIMD vectorization, code generation, and data-aware caching to achieve performance that rivals traditional columnar databases [36]. Photon also has optimizations specifically for data lake patterns, such as aggressive metadata caching to minimize object storage requests, smart file grouping to optimize read patterns, and dynamic partition pruning that can eliminate unnecessary file reads based on runtime information. These improvements, combined with the inherent advantages of open formats like easier data sharing and multi-engine support, have led many organizations to choose data lakes over proprietary storage solutions. The performance gap that once existed between proprietary warehouse formats and open data lake formats has largely disappeared, while data lakes offer greater flexibility and avoid vendor lock-in.

1.4.2 Challenge: New Workloads

As data lake formats have matured to provide the missing functionality of traditional databases while leveraging object storage's reliability and cost advantages, organizations are storing increasingly massive volumes of data in their data lakes, often reaching petabyte scale. This shift has played a crucial role in breaking down traditional data silos within organizations. Historically, different departments or teams within a company might maintain their own separate databases and data warehouses, making it difficult to combine data for cross-functional analytics or machine learning. Data lakes have transformed this landscape by providing a single repository of data from various sources: sales data, customer interactions, product telemetry, or marketing analytics, commonly referred to as the "single source of truth" in industry parlance.

The open nature of these formats means that different teams can use their preferred tools and query engines while working with the same underlying data. For example, data scientists can use Python and Spark for machine learning while business analysts query the same tables using SQL, and data engineers can build ETL pipelines - all without data duplication or complex integration layers. This democratization of data access, combined with the cost-effective storage of object stores and widespread compatibility of data lake formats, hold the promise to allow organizations to have a "single source of truth" for all data.

To fully realize the vision of unified data access, data lakes must expand beyond traditional SQL-based OLAP workloads to support diverse data modalities. Organizations typically maintain multiple specialized OLAP systems: ElasticSearch for text search, vector databases for embedding-based similarity search, and observability platforms like Datadog, Splunk, or Prometheus for metrics, logs, and traces. While these systems have all evolved to embrace the compute-storage separation paradigm common in cloud architectures, their continued existence as separate silos seems unnecessary. Rather than maintaining multiple specialized storage systems, these could potentially exist as purpose-built query engines operating on a shared data lake, where all of an organization's data resides in open formats. This consolidation would not only reduce storage costs and simplify data management, but also enable novel workflows that combine different data modalities - for example, joining SQL tables with vector similarity search results or correlating business metrics with system logs. The key challenge lies in designing storage formats and query interfaces that can efficiently support these diverse analytical patterns while maintaining the performance characteristics that specialized systems currently provide.

In Chapter 4 I will discuss a system, Rottnest, that allows the construction and maintenance of external indices on data lakes. These external indices can then be used to support diverse workloads, such as fast text search or vector search. In Chapter 5 I will showcase a highly optimized Rottnest index, LogCloud, for machine-generated log data, going into depth about the considerations around such external indices.

1.4.3 Contributions Summary

In summary, this dissertation presents four main contributions that advance cloud-native analytical data processing and storage:

Chapter 2: Quokka - Improving Fault Tolerance. We introduce write-ahead lineage, a novel fault tolerance mechanism for dynamic pipelined query engines. Unlike traditional approaches that persist large shuffle partitions (spooling) or expensive state checkpoints, write-ahead lineage logs only kilobytes of task dependency metadata during execution. This approach enables Quokka, our open-source query engine, to achieve 2x speedup over SparkSQL on TPC-H benchmarks while maintaining competitive fault recovery performance through pipeline parallel recovery.

Chapter 3: Leveraging Heterogeneous Clusters. We demonstrate that heterogeneous clusters—combining different VM instance types—can significantly improve cost efficiency for distributed query processing. Using iso-cost curves instead of iso-instance curves, we show that heterogeneous configurations expand the resource constraint polytope, enabling better resource allocation for different query stages, leading to tangible speedups and cost savings in multi-stage joins.

Chapter 4: Rottnest - External Indices for Data Lakes. We present Rottnest, a system for building lazy, object-storage-native indices on data lakes to support diverse search workloads including high-cardinality filtering, substring search, and vector similarity search. Through careful design choices including in-situ indexing and componentization for object storage access patterns, Rottnest overcomes the current shortcomings associated with Parquet-based data lakes for new workloads such as vector and full text search, bridging the gap between expensive dedicated search systems and inefficient brute-force scanning.

Chapter 5: LogCloud - a Rottnest Index for Log Search. Building on Rottnest's framework, we develop LogCloud, a specialized index for machine-generated logs. By combining log-specific compression with a novel object-storage-optimized FM-index implementation, LogCloud provides low search latencies for substring queries with very low total cost of ownership. Evaluation on datasets up to 1.2TB shows LogCloud achieves up to 10x total cost of ownership savings compared to OpenSearch while maintaining interactive search performance using only object storage.

If you would rather skip over the systems details to go straight to the learnings, you could now just jump to **Chapter 6**, where I describe the things I learned about the theory and practice of engineering data systems for the cloud.

Chapter 2

Improving Processing: Fault Tolerance

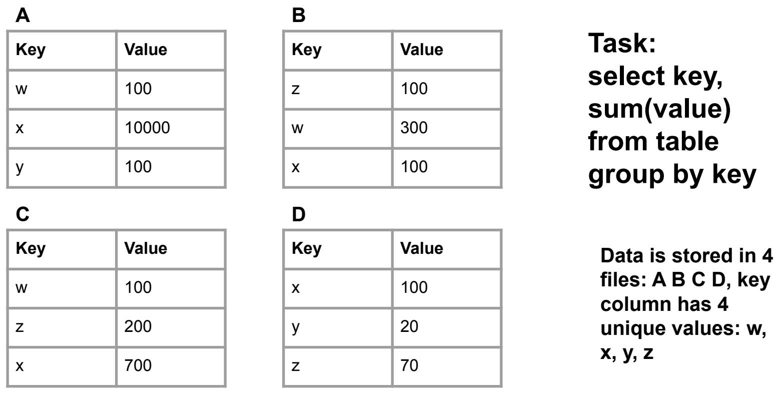
2.1 Motivations

Modern distributed SQL query processing faces a critical tension between performance and fault tolerance. Pipelined distributed query engines like Trino demonstrate significantly better performance than bulk synchronous parallel (BSP) engines like SparkSQL, as we discussed in Chapter 1. However, this performance advantage comes at a cost: these pipelined engines currently lack efficient fault tolerance mechanisms.

While it may be natural for such pipelined engines to borrow fault tolerance mechanisms from streaming systems like Kafka Streams, Apache Flink, and StreamScope [42, 82, 93], which also perform pipelined execution, these mechanisms are actually unsuitable for batch SQL processing. The checkpointing-based strategies used by streaming systems are optimized for continuous processing of small, fresh inputs, and when applied to batch workloads, they introduce significant overhead due to the need to handle large data shuffles.

In this chapter, we will first describe dynamic pipelined execution, the state-of-the-art distributed query engine execution model today, and explain existing fault tolerance strategies and why they fall short. We will then introduce a novel solution to this challenge: **write-ahead lineage**, a fault tolerance strategy specifically designed for pipelined query engines with dynamic task dependencies. Unlike traditional approaches like Spark's static lineage-based recovery [151,152], write-ahead lineage operates by consistently logging lineage information after it has been dynamically determined at runtime, enabling pipelined parallel fault recovery. We demonstrate this approach through Quokka, our open-source distributed query engine implementation.

Write-ahead lineage offers several key advantages. During normal operation, it only requires



A

Key	Value
w	100
x	10000
y	100

B

Key	Value
z	100
w	300
x	100

C

Key	Value
w	100
z	200
x	700

D

Key	Value
x	100
y	20
z	70

Task: select key, sum(value) from table group by key

Data is stored in 4 files: A B C D, key column has 4 unique values: w, x, y, z

Figure 2.1: Example problem we would like to solve.

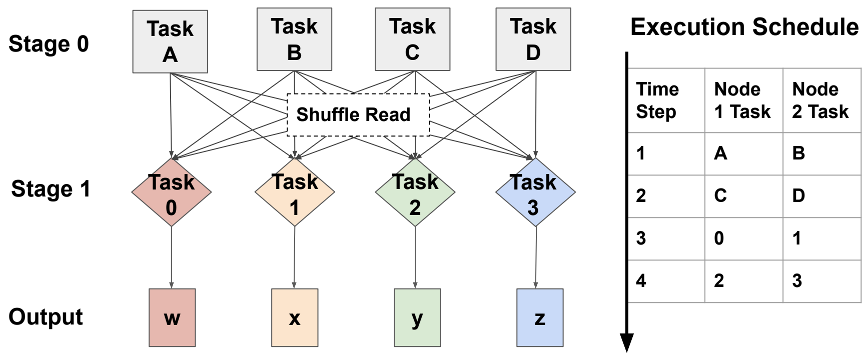
persisting KB-sized lineage information, rather than the MB-sized shuffle partitions or GB-sized state checkpoints needed by traditional approaches. By enforcing a policy where tasks can only consume intermediate outputs whose lineage has been persisted, recovery can be limited to just the tasks scheduled on failed workers. This eliminates the need for the expensive globally coordinated rollbacks common in checkpointing-based systems. Furthermore, our pipelined parallel recovery mechanism enables Quokka to achieve recovery performance comparable to Spark.

Quokka represents, to our knowledge, the first distributed query engine to combine a dynamic pipelined execution model with efficient intra-query fault tolerance. On TPC-H, Quokka outperforms SparkSQL by around 2x on up to 32 nodes in normal execution, with competitive fault recovery performance. While we demonstrate these concepts through Quokka, we believe write-ahead lineage can be readily adapted by other pipelined query engines.

There are decades of related work on pipelined query engines [39, 71, 89, 99, 105, 124, 125] and fault tolerance in dataflow systems [13, 42, 82, 93, 104]. First, let us define what is dynamic pipelined execution in the context of a distributed query engine.

2.2 Background I: What is Dynamic Pipelined Execution?

We will introduce stagewise vs pipelined query engines, then explain static vs. dynamic lineage. To set up our discussion, let us imagine we have to solve the problem shown in Figure 2.1 with a very simple distributed query engine. It proceeds in discrete time steps. We have two nodes in our system. Each can execute 1 task per step. We have a table with two columns, key and value. We would like to perform a simple grouped aggregation on the value column. Let's assume the table is stored across 4 files on object storage, A, B, C and D.



Time Step	Node 1 Task	Node 2 Task
1	A	B
2	C	D
3	0	1
4	2	3

Figure 2.2: Stagewise execution of the example problem with execution schedule shown on the right. The arrow indicates the execution order of the query engine.

2.2.1 Blocking Execution

If we have a stagewise execution engine, the query would execute in two stages, as shown in Figure 2.2. The first stage would have 4 tasks, with each task downloading one of the files. These tasks are labelled **A, B, C and D** after the files they each read. After the tasks read those files, their contents are cached locally on the RAM/disk of the worker node.

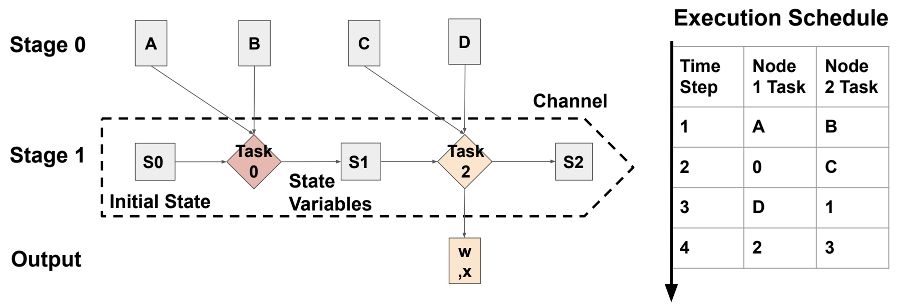
The second stage also has 4 tasks named **0, 1, 2 and 3**, with each task processing all the rows associated with a particular key. In the second stage, task 0 will read all the rows associated with the key value w and sum up their value column to emit the result associated with key = w (500), task 1 will read all the data associated with the key x, etc. Importantly, task 0 needs to read all the rows associated with the key value w across the data of all four input files A, B, C and D.

An example execution schedule is shown in Figure 2.2. In the first two time steps, node 1 and node 2 execute the first stage, downloading A, B, C and D. At time step 3, node 1 and 2 execute tasks 0 and 1. Task 0 requires rows where key = w from the data in input files B and D, which were downloaded on node 2. It will do a networked read, typically called a *shuffle read* in distributed systems literature. Similarly, task 1 will read data from node 1 associated with key = x. In time step 4, the rows corresponding to the other 2 keys are processed.

Popular query engines like MapReduce and Spark follow this paradigm[1]. Two observations:

- The lineage is static. Each task in the second stage must depend on all tasks in the first stage.
- Since tasks in the second stage depend on all the input files, the query engine can only execute tasks of one stage at a time. Since the first stage (time steps 1 and 2) is mostly downloading from object storage, the CPU/RAM on the query engine would be under-utilized. Similarly,

[1]Conceptually. Systems like Spark might have many further optimizations that may appear to deviate from this execution paradigm in practice, like speculative task execution or push-based shuffles.



Time Step	Node 1 Task	Node 2 Task
1	A	B
2	0	C
3	D	1
4	2	3

Figure 2.3: Pipelined execution of the example problem. The dashed box around task 0 and task 2 indicate they are part of the same channel, where task 2 depends on task 0. Task 0 can execute while Task C and D in stage 0 are executing.

in the second stage (time steps 3 and 4), the query engine stops reading from object storage, so the network download bandwidth is under-utilized.

2.2.2 Pipelined Execution

The second observation prompts us to explore ways to start executing tasks in the second stage concurrently with tasks in the first stage to improve resource utilization.

What if instead of having task 0 be responsible for aggregating *all* the rows associated with key = w, we have task 0 only aggregate rows associated with key = w from input files A and B? Then task 0 could start executing as soon as A and B are downloaded, and does not have to wait for C and D. However, now we would have 8 tasks in the second stage. To make things fair with the stagewise approach, we can have task 0 be responsible for the values associated with two keys, w and x. Similarly, task 1 can aggregate the values associated with keys y and z from files A and B. Task 2 can finish aggregating the values associated with w and x for files C and D. We can then achieve the schedule shown on the left side of Figure 2.3. Four observations:

- There is still shuffle read required, since A and B are downloaded on different nodes.
- The number of rows task 0 is responsible for adding is little changed (3 vs 4), as well as the amount of shuffle read required.[2]
- Now task 0 can execute concurrently with reading input C in the same time step! This means the cluster can overlap usage of download bandwidth and CPU/RAM.[3]

[2]We increased task parallelism across input files but decreased parallelism across the key range.

[3]Many problems abound with this statement given our exact toy setup, but if you're acute enough to pick up those problems, you probably also get the general idea. Think about the nodes as two threads on the same machine, and scale up to many machines/threads.

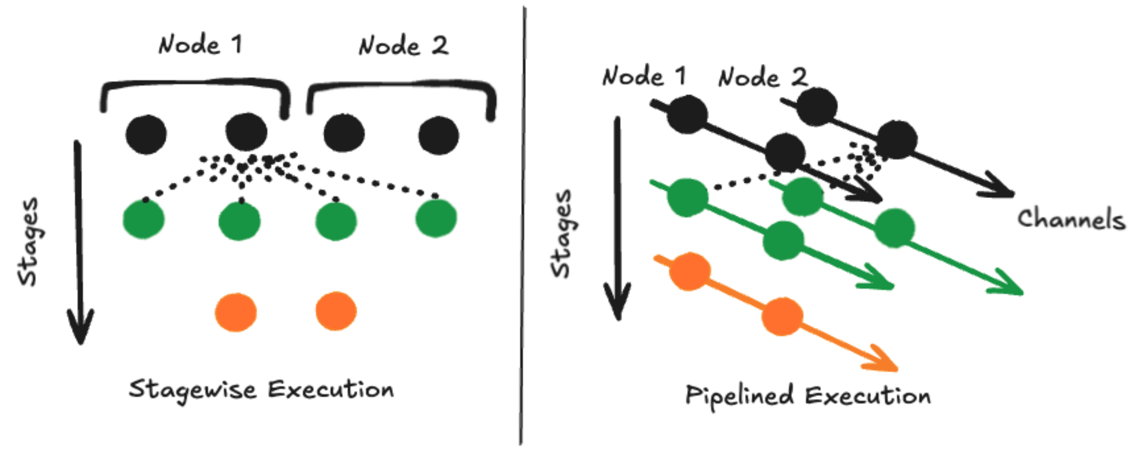


Figure 2.4: Scheduling constraints in stagewise vs. pipelined execution. The dashed lines indicate data dependencies. In pipelined execution, there could still be data dependency across nodes!

- While the schedule still has 4 time steps, since it can achieve better resource utilization in steps 2 and 3 by overlapping download and compute, it will likely execute faster end to end compared to the schedule shown in Figure 2.2.

There is one additional complicating factor, which is that task 0 only computed partial sums of the values associated with keys w and x, and needs task 2 to finish the job! After it finishes the computation, it emits a **state variable**, containing the partial sums, which needs to be consumed by task 2 to emit the final output associated with keys w and x as shown in Figure 2.3.

In this case, the state variable is very small. However, in real problems, this state variable can be massive (i.e. a hash table for a join). If task 2 is scheduled on a different node than task 0, it would have to read the state variable over the network too, which is generally ill-advised.

This limitation introduces a scheduling constraint on the tasks that is not present in the stagewise execution case. For simplicity, we can group tasks that share such state dependencies and thus need to be scheduled on the same node into **channel**s. For this toy example, task 0 and 2 would be such a channel. The second stage consists of two such channels. In our toy example, each channel is scheduled on to their own node. This situation is illustrated in Figure 2.4.

Modern distributed SQL query engines like Snowflake, AWS RedShift, and Trino typically proceed in this pipelined fashion [31, 48, 124].[4] Quokka also adopts this execution paradigm.

Two important observations:

- It is important to note that *channels don't preclude data parallelism.* Even though typically each channel handles a larger range of partition keys in this pipelined paradigm compared to

[4]Again, their implementations may contain major deviations from this simplistic description, but it is a fair mental model for these systems.

tasks in the stagewise parallelism, this is not usually parallelism-limiting, since the ranges of partition keys usually outnumber the amount of parallelism in query engines by a few orders of magnitude.

- A channel is a sequence of tasks. A channel has *only one task executing at a time.* While multiple channels can run in parallel across nodes or or different threads within the same node, within a single channel, tasks execute sequentially. As such, at any time, there are as many tasks executing in the system as there are channels.

  If we take a snapshot of all running tasks in a stagewise query engine, they will all be from the same stage. For a pipelined query engine, such a snapshot will include at most one task from each channel.

2.2.3 Dynamic Lineage

Having covered stagewise vs. pipelined execution, let us explain static vs. dynamic lineage, specifically in the context of pipelined execution. In pipelined execution, each node is typically assigned one of more channels from different stages. Let us assume the perspective of the channel illustrated in Figure 2.3.

1. When the channel starts, it begins with one task, task 0. In our case, the initial state is empty.
2. In our toy example, task 0 is hardcoded to read data from A and B of the first stage, and task 2 is hardcoded to process data from C and D. This means that task 0 need to wait until A and B are ready. Similarly, when task 0 is finished and assuming C and D are not produced, task 2 cannot start.
3. Instead of hardcoding task depedencies between tasks of one channel and those of an upstream channel, we can let the channel decide for itself. For example, when the node finishes executing task 0 of the channel and only the data for C is present, task 1 of this channel could just kick off with C. This is referred to as **dynamic lineage**. The channel could make the decision based on heuristics or some user-defined policy.[5]
4. This implies that in a pipelined system with dynamic lineage, the total number of tasks per channel is not fixed. Instead a channel terminates when it no longer receives eligible inputs to schedule for the next task.

More recent distributed query engines such as the newest Amazon RedShift version determine task adopt dynamic lineage [31], inspired by multicore database innovations such as work-stealing and morsel-driven parallelism [89]. We shall henceforth refer to pipelined query engines with dynamic lineage as **dynamic pipelined query engines**.

[5]This is a form of distributed task scheduling. Similar to how distributed routing protocols can alleviate network congestion, good task scheduling policies by channels can make the data go through the query engine faster.

Table 2.1: Fault tolerance design choices in data processing systems.

	Trino	**SparkSQL**	**Kafka Streams**	**Flink**	**StreamScope**	**Quokka**
Description	Pipelined SQL	Stagewise SQL	Dataflow	Dataflow	Dataflow	Pipelined SQL
Spooling	✓	✗	✓	✗	✗	✗
State Check-point	✗	✗	✓	✓	✓	✗
Lineage	✓	✓	✓	✗	✓	✓

2.3 Background II: Fault Tolerance Challenges

The principal question we aim to answer in this chapter is: *how to support fault tolerance for dynamic pipelined query engines with low overhead and fast recovery*? As discussed, pipelined query egnines consist of a collection of channels scheduled across different nodes. When a node dies, some of those channels may fail and need to be recovered.

In this section, we explore existing approaches and describe their weaknesses.

Most distributed pipelined query engines do not support intra-query fault tolerance, instead relying on query-retries when a cloud worker instance fails [31, 48]. Only Trino recently added support for intra-query fault tolerance based on HDFS spooling of shuffle partitions [7].

While the literature on intra-query fault tolerance has been relatively meager, we should also mention stream processing systems based on the dataflow model [104]. These systems also often consist of a collection of stateful actors scheduled across different nodes, which differ from pipelined query engines mainly in the granularity of the data that is passed between different tasks. Fault tolerance for such streaming systems has been thoroughly studied in the past decade, with strategies coalescing around three core techniques, similar to the taxonomy proposed by Falkirk Wheel [64]:

- **Lineage**: Lineage refers to the dependencies between data partitions. In systems with dynamic lineage, the lineage can be persisted to be consulted upon failure to facilitate recovery.[6] For example, in Figure 2.5, we would write down that task 2 depends on input C and D.
- **Spooling vs Upstream Backup**: Input data to tasks, like C and D in Figure 2.5, may be stored reliably (*spooling*), unreliably, e.g. on local disk of the producer (*upstream backup*), or not at all.
- **Checkpointing**: state variables, like S1 in Figure 2.5, can be persisted.

Table 2.1 summarizes the fault tolerant systems we compare in this section, along with which of the three strategies they employ. Trino and SparkSQL are distributed query engines, while Kafka

[6]In systems with static lineage, this information is trivially known before execution, so no persisting is required.

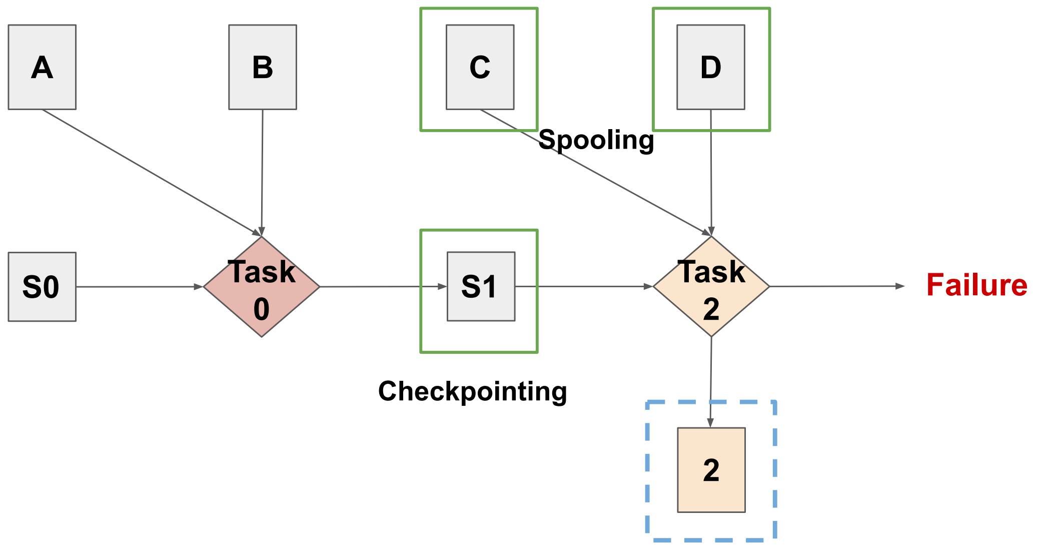


Figure 2.5: The inputs to task 2 are marked in green boxes, while its output is in blue dashed box. Spooling would mean persisting C and D, whereas checkpointing would persist S1.

Streams, Flink and StreamScope are stream processing engines. We explain how each of the three strategies are applied in these five systems.

2.3.1 Lineage

Tracking lineage allows a task to guarantee a fixed output by remembering what inputs were used, assuming the task is deterministic.

Of all the systems, Flink is the only one which does not track lineage [42]. Upon fault recovery, failed tasks that need to be relaunched can use different inputs the second time around [4]. Critically, this decision means tasks could emit different outputs and workers who previously consumed the outputs of the failed task also have to be rewound, which typically results in expensive coordinated global rollbacks of all the channels in the entire system, even the channels that did not fail!

All the other systems listed in Table 1 determine lineage statically. Trino and Spark determine task dependencies before the query graph is executed [7, 152], while real-time systems such as Kafka Streams and StreamScope rely on the unique event time associated with each record to impose a deterministic execution order among inputs [82, 93]. In either case, this lineage is assumed to be available after a worker fails.

2.3.2 Upstream Backup / Spooling

If we decide to track lineage, making use of it upon fault recovery requires some way of replaying a task's inputs using the lineage, which typically requires storing intermediate data partitions.

MapReduce pioneered this approach by persisting reducer outputs in GFS to provide fault tolerance boundaries between different stages [51]. Trino stores intermediate data partitions durably in HDFS or an object storage like S3, while Kafka Streams persists them in Kafka topics [7, 82].

However, persisting data partitions, a.k.a spooling, can introduce severe overheads in normal operation, especially in batch analytics. Persisting a data partition in a distributed cluster where workers might fail means either replicating the data partition across the cluster, e.g. Kafka topic or HDFS, or writing the data partition to a blob storage, e.g. Amazon S3. In either case, this operation consumes precious network I/O resources that could be used for the task itself.

In contrast, Spark relies on unreliable upstream backup of data partitions to local disk of the producer, which is assumed to be lost upon worker failure. Instance-attached NVMe drives have become ubiquitous on public cloud providers, making writing to local disk very efficient compared to persistent writes to HDFS or S3, though the contents of such drives are lost upon worker failure. Avoiding spooling is a key reason why Spark is faster than MapReduce [152].

A bigger problem for spooling in a pipelined engine is that it might not save that much work upon failure. The core benefit of spooling is the localization of task retries. In a system relying on upstream backup, if the input to a task that must be retried is also lost, then the task that generated that input must also be retried. Spooling avoids this problem, but only if all of the inputs for the failed task have been persisted.

Unfortunately, in pipelined query engines, tasks also depend on the channel's state variable. In Figure 2.5, we illustrate what data partitions are persisted in a typical spooling strategy. If we only persisted data partitions C and D and the channel experiences a failure after executing task 2, it cannot skip re-executing task 0 because task 2 depends on the state variable S1. Streaming engines that perform spooling also commonly "checkpoint" these state variables.

2.3.3 Checkpointing

We could prevent restarting the failed channel from scratch if we periodically persist the lost state variables. Checkpointing executor state periodically is a popular fault tolerance strategy in real-time streaming systems such as Apache Flink, Kafka Streams and StreamScope. Since jobs in these systems could be continuously operating for days, restarting a channel entirely might cause unacceptable fault recovery performance. Checkpointing also allows the system to garbage collect spooled data partitions, since a data partition does not have to be replayed if its effect has been persisted into a state checkpoint.

However, checkpointing can be even more expensive than spooling for pipelined query engines optimized for SQL queries on large batches of data. Streaming systems typically go to great lengths to ensure that the state of an operator is bounded in size. SQL query engines have no such requirements: consider an operator that builds a hash table for joins. The size of the hash table grows linearly with the number of unique keys it sees. Assuming new keys arrive at a constant rate, naive periodic checkpointing will incur $O(N^2)$ storage complexity where N is the number of unique keys, which can be very large.

Incremental checkpointing could be employed to checkpoint only differences between adjacent checkpoints. While we can easily devise incremental checkpointing strategies for individual stateful operators, efficient generic incremental checkpointing strategies for arbitrary data structures is still an open research problem. Current approaches include persisting a "changelog" of the state as in Kafka Streams or leveraging RocksDB's compaction mechanism as in Apache Flink [42, 82]. Both impose heavy constraints on the underlying data structures of the state variable, which is not desirable for a high performance query engine.

2.3.4 Priorities

The key issue is that techniques like spooling and checkpointing are developed for streaming systems where the priority in fault recovery is fast recovery latency. This is relevant for streaming systems powering critical real-time services like fraud detection or trading systems, but less relevant for distributed query engines.

In the context of pipelined query execution, fault tolerance's first priority should be low overhead. If we cannot achieve low overhead in normal pipelined execution, we are better off running without fault tolerance and retrying queries that fail or using blocking alternatives such as SparkSQL. This analysis naturally leads us to rule out considering spooling and checkpointing.

Having satisfied low overhead, we can accept high fault recovery latency as long as we have good fault recovery *throughput*. It is acceptable to recompute a lot of data partitions, as long as we can do so in parallel by leveraging the ample amount of parallelism typically present in a distributed query engine. This philosophy is similar to that of Spark, which chose to store data in memory instead of disk to lower overhead in normal execution [151].

2.4 Write-ahead Lineage

We propose a simple fault tolerance strategy for distributed query engines called **write-ahead lineage**. Shown in Algorithm 1, the strategy simple: *as tasks process data partitions, the dynamically determined lineage is persisted to a transactional data store, the GCS*[7]*, before the output can be consumed by downstream tasks.* Importantly, *tasks can only consume data partitions with committed lineage.* This means that in the example shown in Figure 2.5, a downstream task can only consume data partition 2 marked in the blue dashed box when task 2 has persisted its lineage to the GCS. When a task fails, this logged lineage can be consulted to recover from the failure by replaying data partitions and retrying tasks, similar to how Spark uses its statically determined lineage for recovery and how ACID databases recover data from the write-ahead log [151].

[7]Global Control Store, inspired by Ray

Algorithm 1 Write-ahead Lineage

```
Given task $\tau$ on worker $\omega$, with GCS $\mathcal{G}$, where $\mathcal{G}.\mathcal{L}$ stores
committed lineages, $\mathcal{G}.\mathcal{T}$ stores outstanding tasks
$\mathcal{A} \leftarrow$ all data partitions pushed to $\omega$
$\mathcal{B} \leftarrow$ all possible inputs to $\tau$
$\mathcal{I} \leftarrow \{x \in \mathcal{A} \cap \mathcal{B} \mid x \in \mathcal{G}.\mathcal{L}\}$
if $\mathcal{I} = \emptyset$ then
    return                                    ▷ No inputs with committed lineage available
end if
Execute $\tau$, push results downstream
Store results locally on disk (upstream backup)
if push results failed then
    return                                    ▷ Downstream worker failure, do not commit
end if
Set $\tau$ to $\mathcal{I}$ in $\mathcal{G}.\mathcal{L}$, remove $\tau$ from $\mathcal{G}.\mathcal{T}$ in a single transaction.
return                                        ▷ Success
```

Intuitively, write-ahead lineage upholds the core invariant of lineage-based recovery: *tasks consume only objects with committed lineage*. This ensures that a task's output stays the same after failure recovery, so channels that did not suffer failures do not have to be rewound. There are two classes of channels to consider here: those whose output is consumed by the failed task and those who consume the output of the failed task. For the first class, because all past outputs are backed up to local disk, outputs can be replayed. Channels in the second class simply ignore the recovered task's re-transmitted output until the failed channel has recovered to the state before failure.[8]

There are two ways to enforce the core invariant: check lineage before consuming inputs or commit lineage before pushing outputs. Quokka adopts the former approach to minimize write transactions to the GCS, which lets the lineage be written as the last step of the algorithm. Quokka can then bundle this write with other writes to the GCS, like removing τ from the task queue and adding the next task in the channel, as a single transaction.

Only eventual consistency is required for the lineage. If a task does not immediately see a required input's lineage in the GCS, it will simply exit without being executed. The task will be tried again later and successfully execute when the lineage becomes visible.

[8]In systems based on Chandy-Lamport without lineage tracking, a failed worker can process inputs in different orders upon recovery, causing it to retransmit different messages than before failure. In Falkirk Wheel, this is called the "no messages are duplicated" constraint. This typically results in expensive coordinated rollbacks of all channels to a globally consistent state in these systems, which we avoid [42, 64].

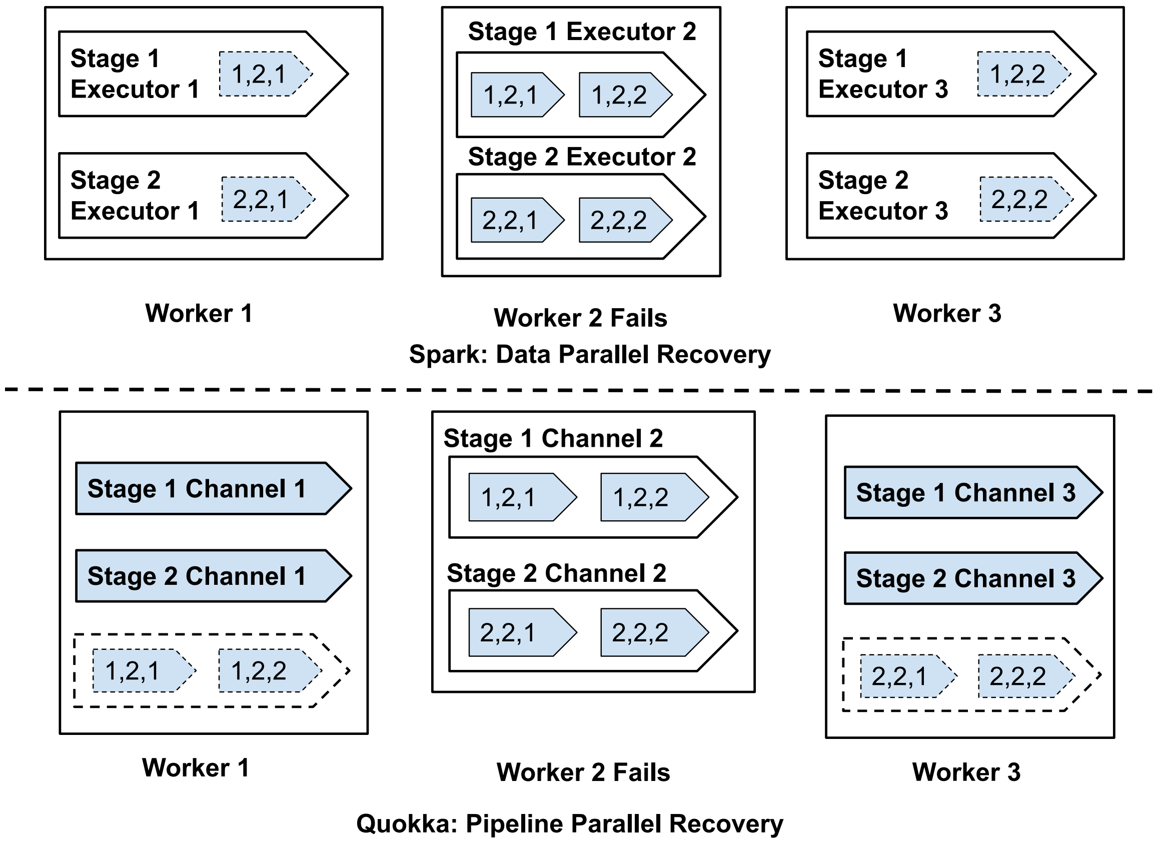


Figure 2.6: We show a job with 3 workers and 2 sequential stages. We assume worker 2 fails, and we have to reconstruct the outputs of its 4 tasks across two stages. In the case of *data parallel recovery*, each surviving worker would handle 1 task of each stage. For *pipelined parallel recovery*, each surviving worker would handle tasks of an entire stage. The effective parallelism achieved is similar in this case.

2.4.1 Lineage Naming Scheme

Recording lineage requires a naming scheme for tasks and their outputs. In Quokka, we introduce a naming scheme that allows a very succinct representation of the lineage to minimize logging overhead. The name of a task is a tuple of the form *(stage, channel, sequence number)*. The sequence number increases monotonically within each channel. A task's output has the same name as the task. Tasks must consume their inputs in order. As an example, in a query with two stages and two channels in each stage, a task in channel 1 in stage 2 could depend on outputs from tasks in either channel in stage 1. However, it must consume output from task (1,1,0) before task (1,1,1). In Quokka, we further restrict tasks to consume from one upstream channel at a time. A task decides at runtime how many task outputs from that channel to consume.

Under this execution model, a task's input requirement $\mathcal{B}$ can be described as a vector of length C, where C is the number of upstream channels it could depend on. The i^{th} element denotes the number of consumed outputs from channel i similar to a watermark. A task's lineage can be described with just two numbers, i and K, the upstream channel it consumed from and how many outputs it consumed (recall we limit tasks to consume from one channel at a time in Quokka). This is much less information to log than a naive scheme where we assign unique names to all outputs in the system and track all input names for each task.

Our experiments indicate that write-ahead lineage's overhead in normal operation typically results from the disk writes needed for upstream backup.[9] Quokka's upstream backup is similar to Spark, with the key difference being that in Quokka, multiple stages could be writing shuffle partitions at the same time, exerting higher disk pressure than Spark. We believe this design is reasonable as fast instance-attached NVMe SSDs are becoming more popular on public clouds, and network throughput used in shuffling data partitions will be saturated before disk throughput used in backing them up. The total amount of data stored for the entire job is the same between Quokka and Spark, as Spark typically maintains shuffle partitions of all stages.

2.4.2 Pipeline Parallel Recovery

When a worker fails, the channels scheduled on it will lose their current active tasks, the associated state variables, and some cached data partitions. Quokka attempts to recover these channels to their previous state before failure by reconstructing lost partitions and state based on the logged lineage.

Tasks in channels that do not contain state variables, typically input readers from object storage or stateless user defined functions, can be recovered in parallel across the cluster similar to Spark. However, within a failed channel with state variables, tasks must be reconstructed in sequence.

Even though tasks must be reconstructed in sequence *within* a stateful channel, Quokka can still accomplish parallel recovery *between* channels, as illustrated in Figure 2.6. In Quokka, if the query

[9] Recall upstream backup means storing task outputs in ephemeral storage instead of persisting them (spooling).

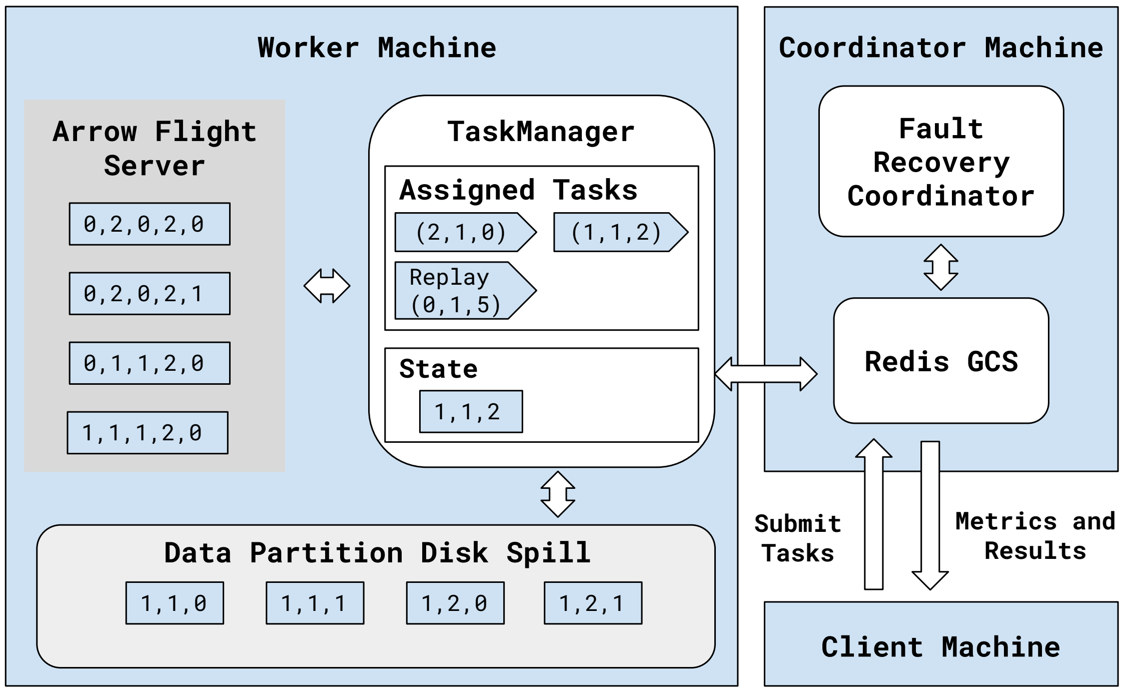


Figure 2.7: Quokka's architecture. Note that instead of having components communicate with each other through RPC calls, all coordination is done through the GCS. The client also communicates with the cluster through the GCS.

contains multiple stages (e.g. a multi-way join), a failed worker contains many stateful channels that need to be reconstructed. These stateful channels belonging to different stages can be scheduled on different workers, in a pipelined parallel fashion.

Compared to Spark where the degree of parallelism is proportional to the number of tasks per node in a stage, the degree of parallelism here is proportional to the number of pipelined stages in the query. In production workloads, we find these two numbers to often be quite similar.

2.5 Implementation

We now describe how write-ahead lineage fits with other pieces of the query engine in Quokka. A simplified schematic of Quokka's architecture is shown in Figure 2.7.

2.5.1 Architecture

Quokka uses a cluster of worker machines, which might fail at any time, e.g. due to spot instance pre-emptions or Kubernetes pod evictions. Quokka also needs a head node, which could be one of the worker machines, or a separate instance. Like Spark, Quokka assumes the head node does not fail [152], which can be achieved by using on-demand instances that cannot be preempted, or Kubernetes scheduling policies. In addition, we assume there is a client machine, which could be the user's laptop or another cloud instance, that submits jobs to this cluster.

TaskManagers

Quokka is implemented on top of Ray, a Python-based actor framework [103]. Each physical worker node is assigned a TaskManagers similar to Apache Flink [42]. As discussed, in Quokka each node is assigned a collection of channels, each of which has one active task at a time. The TaskManager is in charge of scheduling the next active task(s) to run on that worker node. For example, the TaskManager shown in Figure 2.7 is assigned channel 1 from stages 1 and 2. It can decide between active tasks (2,1,0) (the first task in cahnnel 1 for stage 2) and (1,1,2).

Task dependencies in Quokka are determined dynamically at runtime by the TaskManager, in a distributed fashion without coordination. Each TaskManager looks at its available tasks and input data partitions and decide based on heuristics or some user-defined policy. We believe that studying intelligent task scheduling strategies is a promising future research area.

While Quokka supports multiple scheduling strategies, the results presented in this Chapter adopts a simple strategy: each task executes as soon as there is any available inputs. Some of Quokka's single-node kernels (i.e. joins with Polars/DuckDB) might prefer larger batch sizes, so this strategy might not be optimal for them. In these cases, Quokka might give up a bit of single task efficiency but would retain the benefit of lower pipeline latency. The strategy is also self-healing. When a TaskManager executes a task with very little data, it gives upstream channels more time to buffer up outputs, leading to the next task being more efficiently executed on more data.

Push-based

Quokka employs a push-based query execution model where producer tasks actively send their outputs directly to the TaskManagers hosting consumer tasks. The system operates on a static channel-to-TaskManager mapping, enabling producers to determine the destination TaskManager for their outputs at runtime.

A key feature of Quokka is its support for dynamic lineage—tasks can decide which upstream outputs to batch together during execution rather than having this predetermined. While this flexibility exists at the task level, the channel routing remains deterministic: producer tasks always know which channels should receive their outputs, even though they may not know the specific task within that channel that will ultimately consume the data.

All the TaskManagers on the same machine share access to an Apache Arrow Flight server, which manages zero-serialization data communication between different machines [1]. In Quokka a task pushes its outputs directly to the Arrow Flight servers of all its downstream consumer channels.

We found the performance of this approach exceeds moving data via Ray's built-in object store and offers more flexibility in terms of handling disk spilling and chunking shuffle batches.

In addition to sending the task output downstream, the TaskManager would store the task output locally in case it needs to be replayed. All TaskManagers on the same machine share access to the instance-attached disk. Since the naming scheme described in Section 2.4.1 ensures that the task outputs from different channels are named differently, there is no need for synchronization among different TaskManagers for writes.

2.5.2 Coordination through Transactions

The write-ahead lineage algorithm requires a persistent transactional data store, which we call the GCS. Quokka uses a Redis server on the head node to implement the GCS. Since we assume the head node does not fail during the job, anything logged to the Redis server is considered "persisted". If head node failure is a concern, DynamoDB can be used instead.

The use of a GCS is inspired by the design of modern distributed systems like Kubernetes and Ray that offload the control plane to a data store such as etcd or Redis [40, 103]. In addition to the lineage, the GCS holds the single source of truth for the execution state of the entire system in Quokka, such as the tasks assigned to each TaskManager and what data partitions are present on which machines. Individual TaskManagers are stateless and actively poll the GCS for tasks assigned to them to execute the write-ahead lineage algorithm in Algorithm 1.

The coordinator periodically polls TaskManagers to see if any of them have failed. Once a failure is detected, the coordinator sets a control flag in the GCS. TaskManagers periodically poll this flag. If they see the flag is set, they abort their current tasks and wait. This barrier effectively implements a GCS-level lock to guarantee the coordinator exclusive read-write access to the GCS without potential conflicts. The coordinator then proceeds to schedule pipelined parallel recovery of tasks as described in Section 2.4.2.

The usage of a centralized GCS greatly simplifies the implementation of the coordinator. The coordinator's fault recovery routine simply updates the GCS transactionally with the tasks that need to be retried instead of interacting directly with TaskManagers. We rely on the strong read-after-write consistency properties of the GCS to ensure that all TaskManagers have the latest correct view of the updated pending tasks after the TaskManagers restart.

This separation provides us the key advantage that the coordinator does not have to assume the remaining TaskManagers are all alive, simplifying the handling of nested failures. We found this design to be much simpler and less error-prone than a traditional approach where different components of the system interact through RPC calls between the coordinator and TaskManagers, and allows us to avoid some SparkSQL fault recovery problems we encounter in practice on AWS EMR described in Section 2.6.4.

A potential trade-off of a centralized GCS is performance and GCS memory footprint, especially when every task has to write to the GCS. Ray now uses the concept of Ownership to conduct distributed lineage tracking to reduce lineage logging and storage overhead [141], where a task commits its lineage to other tasks who have a claim to its results. However, with the optimized task naming scheme described in Section 2.4.1, both the GCS logging overhead and its memory footprint become negligible in Quokka.

One might question the scalability of a design that relies on a centralized GCS on the hot path of task processing. Running Redis benchmark on the head node used in our experiments yielded a throughput of 75K RPS for 10KB-payload SET operations with 1000 concurrent clients. Assuming each task processes 40MB of data for each GCS operation, this caps the data processing throughput across all stages at 3TB/s. Assuming five stages, the input throughput is limited to 600GB/s, which corresponds to 480 worker machines at 10Gbps download throughput per machine, which is much larger than the typical cluster sizes used in practice [138]. If Redis on the head node becomes a bottleneck, it can be replaced by a Redis cluster or DynamoDB.

2.5.3 Failure Recovery

We have already described Quokka's pipeline parallel recovery at a high level in Section 2.4.2. Here we walk through its implementation in Quokka's architecture.

The failure recovery implementation is motivated by Kubernetes' philosophy of *reconciliation*. When a failure occurs, the GCS now contains inconsistent information, such as tasks assigned to failed workers. During fault recovery, the coordinator updates it to a consistent state satisfying the following constraints:

- Lost tasks are rescheduled on live TaskManagers.
- All the input data partitions needed for any existing or rescheduled task will be replayed or recomputed.

The algorithm used by the coordinator in Quokka is shown in Algorithm 2 with simplifications[10]. It implements pipeline parallel recovery described in Section 2.4.2 by assigning different rewound stateful channels to different workers.

A concrete example is shown in Figure 2.9 for a Quokka application with three stages. Stage 0 is stateless and stage 1 and stage 2 make use of state variables. After a TaskManager fails, the coordinator will first reschedule all its tasks, (1,2,1) and (2,2,1) in this example, to other live TaskManagers. The channels associated with the failed tasks are restarted from the initial state, so the tasks that need to be launched are (1,2,0) and (2,2,0), which can be relaunched on different machines.

The coordinator now traverses the stages in reverse topological order and checks if the required data partitions for the relaunched tasks are still present. If so, e.g. (0,0,0), (0,1,0) ... , replay tasks

[10]In Algorithm 2, we assume there are no lost replay or input tasks, though Quokka handles those as well.

Algorithm 2 Failure Recovery Algorithm

```
Assume GCS $\mathcal{G}$, where $\mathcal{G}.\mathcal{L}$ stores committed lineages, $\mathcal{G}.\mathcal{T}$ stores outstanding tasks
$\mathcal{X} \leftarrow$ the set of all tasks assigned to the failed worker
$\mathcal{R} \leftarrow \{(\tau.\text{stage}, \tau.\text{channel}) \mid \tau \in \mathcal{X}\}$ (Set of rewind requests)
for each $(stage, channel)$ in reverse topological order do
    if $(stage, channel) \in \mathcal{R}$ then
        Identify required inputs $\mathcal{I}$ for $(stage, channel)$ from lineages of $channel$ outputs in $\mathcal{G}.\mathcal{L}$
        for each data partition $(stage, channel, seq)$ in $\mathcal{I}$ do
            If exists, add replay task to the owner worker
            Else if $stage$ is input, add input task to any node
            Else, add $(stage, channel, 0)$ to $\mathcal{R}$
        end for
    end if
end for
for each $(stage, channel, 0)$ in $\mathcal{R}$ do
    Remove $(stage, channel, seq)$ from $\mathcal{G}.\mathcal{T}$
    Assign $(stage, channel, 0)$ to a random worker in $\mathcal{G}.\mathcal{T}$
end for
```

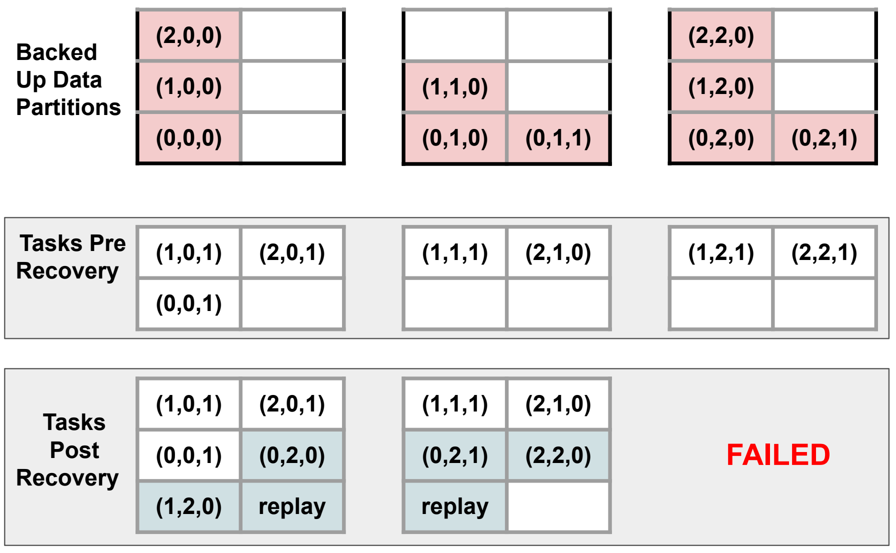


Figure 2.8: An example fault recovery procedure when one out of three workers fail. Pink shade represents data partitions that have been generated by past tasks and stored on the TaskManager. Recovery tasks are shaded in light blue.

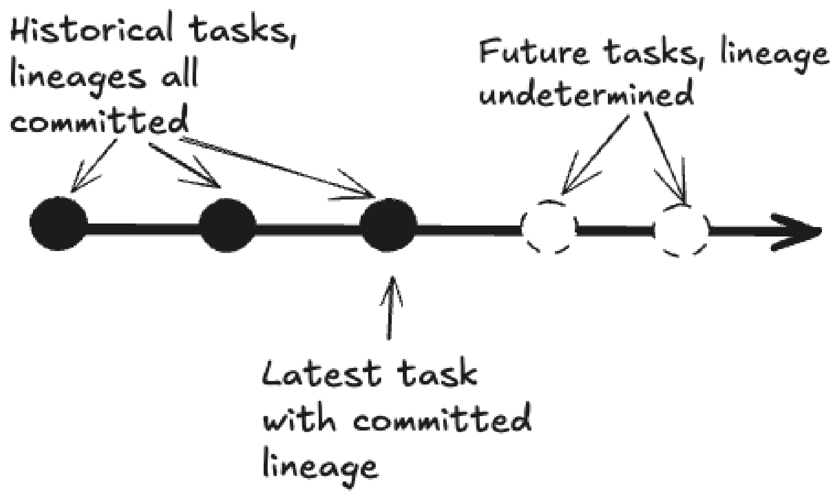

Historical tasks, lineages all committed
Future tasks, lineage undetermined
Latest task with committed lineage

Figure 2.9: A channel's execution timeline showing a strict ordering: historical tasks (filled circles) with committed lineages always precede future tasks (empty circles) with undetermined lineages.

are pushed to TaskManagers that hold them. If not, the data partition must be regenerated by rewinding other channels. This is typically due to tasks on the failed machine depending on data partitions held by the same machine, i.e. (0,2,0) and (0,2,1).

As mentioned in Section 2.4, Quokka can engage in pipelined parallel recovery between channels, so (1,2,0) and (2,2,0) can be rescheduled to be recovered on different workers. Quokka can also parallelize recovery between tasks in the same channel if there is no state dependencies involved. For example, (0,2,0) and (0,2,1) can be rescheduled on different workers. In total four data partitions need to be reconstructed in this failure scenario, which corresponds to the total number of data partitions stored on the failed machine.

When a rewound task (such as (1,2,0) or (2,2,0)) retraces its execution path, it loses the freedom to dynamically select input data partitions. Instead, it must consult the Global Control Store (GCS) to retrieve the exact lineage required to regenerate each output partition. This constraint ensures that rewound channels produce identical outputs to their pre-failure state.

From a conceptual perspective, each channel's execution history forms a sequence of naturally numbered tasks. This sequence comprises two distinct categories: historical tasks with committed lineages and future tasks with undetermined lineages. The system can rewind a channel multiple times while preserving correctness by adhering to a simple principle: follow the persisted lineage for any task whose execution history has been recorded. Only when encountering a task without persisted lineage can the channel resume dynamic decision-making about its execution path.

This design elegantly separates deterministic recovery from dynamic execution – tasks bound by history must faithfully reproduce their past behavior, while tasks venturing into uncharted territory

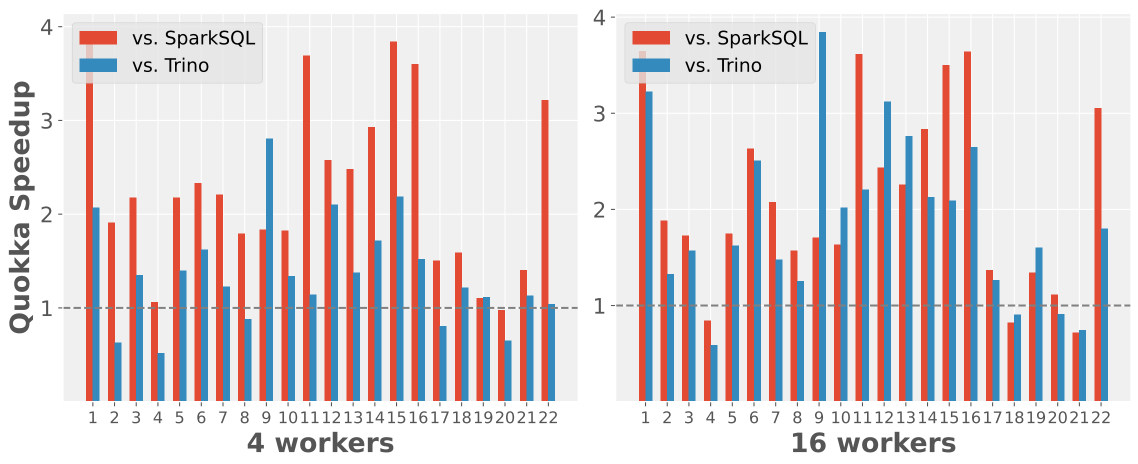


Figure 2.10: Comparing the performance of fault-tolerant data processing systems (Trino with FT, SparkSQL and Quokka) on the TPC-H queries (query number indicated on x-axis) on a) 4-worker cluster and b) 16-worker cluster. Quokka outperforms Trino and SparkSQL in most cases.

retain full autonomy over deciding their inputs.

2.6 Evaluation

We test Quokka's performance, fault tolerance and scalability on the full TPC-H benchmark (scale factor 100) with input in Parquet format stored on AWS S3. We then select 8 representative queries in three different categories:

- **I**: simple aggregations (1, 6)
- **II**: simple pipelined joins (3, 10)
- **III**: queries with multiple join pipelines (5, 7, 8, 9)

We perform detailed ablation studies of different design choices and measure fault recovery performance on these representative queries. These queries are chosen because they contain mostly just one join tree, reducing confounding variables on performance.

Quokka is run on a Ray cluster with Ray version 2.4 on AWS EC2 on-demand instances. We use two cluster configurations. The first configuration uses four r6id.2xlarge worker machines. The second uses 16 r6id.xlarge machines. An r6id.2xlarge instance has 8 vCPUs, 64GB of RAM ad 474GB of instance attached NVMe SSD. An r6id.xlarge instance has exactly half of those resources.

For comparison, we benchmark SparkSQL 3.3 and Trino 398 on AWS EMR 6.9.0 on the same cluster configurations. AWS EMR configures Spark to also use NVMe SSDs for potential spilling.

We further optimize the network and shuffle retry configurations of SparkSQL to start fault recovery in two seconds, instead of the default two minutes on AWS EMR, to match the behavior of Quokka. Trino is benchmarked with and without fault tolerance by HDFS spooling. SparkSQL is fault tolerant by default. Before running the queries, `ANALYZE` commands are run for both SparkSQL and Trino to ensure cardinality-based optimizations are enabled.

Unless otherwise noted, all timing results are from the mean of three independent measurements, with standard deviation shown as error bars when applicable.

2.6.1 Quokka vs. Trino vs. SparkSQL

In Figure 2.10, we compare Quokka's performance to Trino with spooling-based fault tolerance and SparkSQL on the full TPC-H benchmark. We see that for most queries across both cluster configurations, Quokka is the most performant among all three query engines. Compared to Trino, Quokka achieves 25% geometric mean speedup on the 4-worker cluster and 70% on the 16-worker cluster. Compared to SparkSQL, Quokka achieves 2.1x geometric mean speedup on the 4-worker cluster and 1.9x on the 16-worker cluster.

It is important to note that a lot of factors could contribute to these results. All three systems employ different kernels to implement SQL operators such as join and filter, libraries for networked communication and task scheduling systems. However, these results do indicate that Quokka's implementation is competitive with state-of-the-art data processing systems. We hypothesize Quokka's speedup over Spark is mostly due to blocking vs pipelined execution and Quokka's speedup over Trino is due to Trino's high spooling overhead.

We note that Quokka's performance against SparkSQL and Trino is worst for complicated queries that contain nested subqueries and might require materialization of intermediate results (e.g. 2, 4, 20, 21) and better for simpler queries like 8, 9 or 12 that contain one join tree. The reason for this discrepancy is Quokka currently must perform expensive global synchronization between pipelines, making complicated queries that contain multiple pipelines slow. In addition, Quokka's implementations of semi-joins and anti-joins, required to unnest subqueries, are not yet very efficient.

Quokka's advantage against SparkSQL and Trino is maintained on the 16-worker cluster compared to the 4-worker cluster, suggesting that Quokka's design is scalable to larger cluster sizes.

2.6.2 Why Dynamic Pipelined Execution?

We now show that dynamic pipelined query execution leads to significant performance gains compared to both stagewise execution and pipelined query execution with static task dependencies to motivate the need for a fault tolerance algorithm specifically designed for dynamic pipelined query engines.

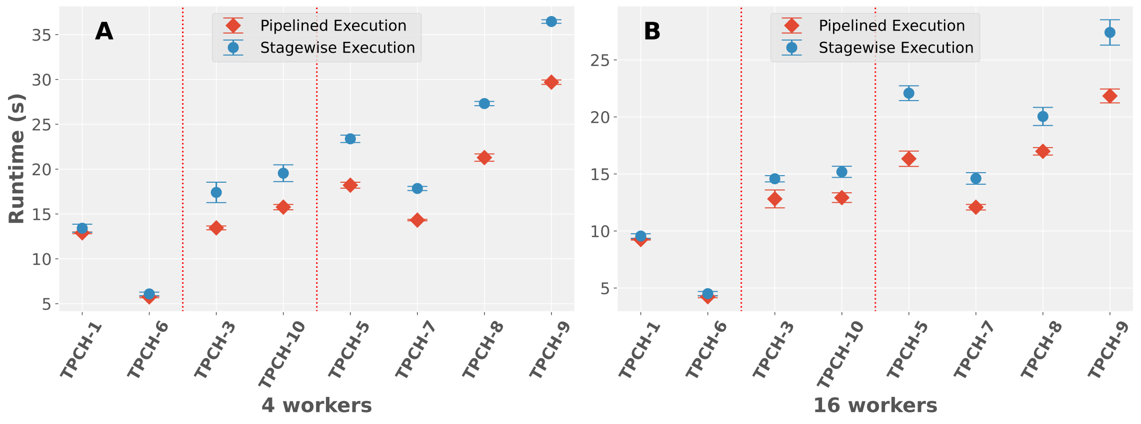


Figure 2.11: Pipelined Quokka vs Stagewise (blocking) Quokka execution times on the TPC-H queries on the a) 4-worker cluster and b) 16-worker cluster. Pipelined execution outperforms in all cases.

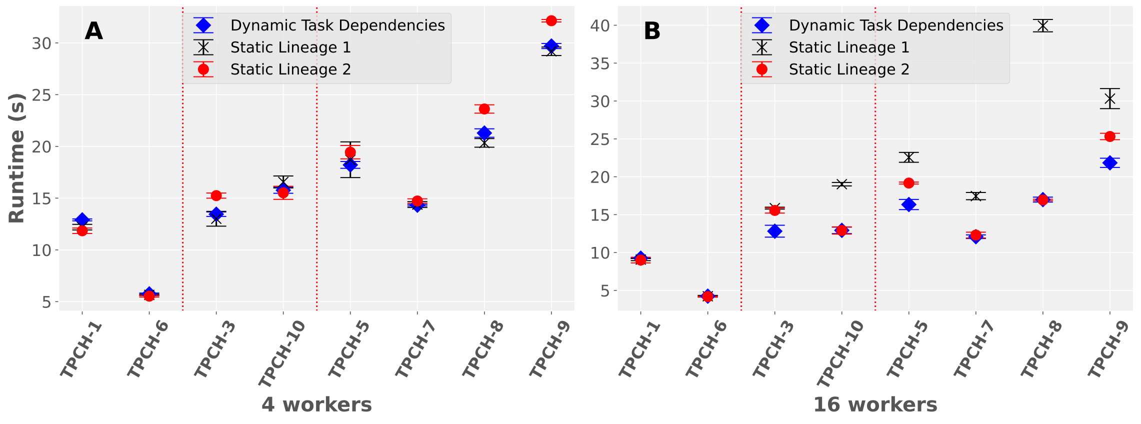


Figure 2.12: Performance of Quokka with dynamic task dependencies vs. two different static lineage strategies on the a) 4-worker cluster and b) 16-worker cluster. Strategy 1 (batch size 8) outperforms strategy 2 (batch size 128) on the 4-worker cluster but greatly underperforms on the 16-worker cluster. Enabling dynamic task dependencies allows Quokka to match the better performing static strategy in most cases.

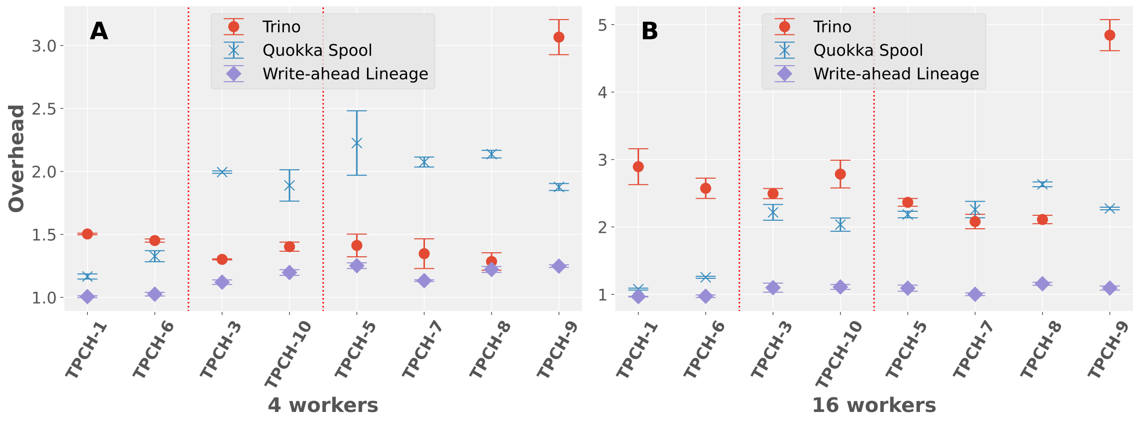


Figure 2.13: Trino's HDFS spooling fault tolerance overhead, Quokka S3 spooling overhead and write-ahead lineage overhead on the a) 4-worker cluster and b) 16-worker cluster. Overhead of 1 means no overhead.

Pipelined vs Blocking Execution

We modify Quokka to execute in a stage-wise fashion similar to SparkSQL and examine its performance degradation. Results for both cluster sizes are shown in Figure 2.11.

Across the eight selected queries across the two different cluster setups pipelined execution consistently outperforms stagewise execution. The speedups are especially significant for queries in category III, which involve multiple joins that can be pipelined. On queries 1 and 6 in category I where the query consists of essentially just the read stage, the pipelined execution does not improve runtime, as expected.

Overall, pipelined execution leads to 26% geometric mean speedup on the queries in categories II and III on the 4-worker cluster and 22% speedup on the 16-worker cluster. On queries with deep join trees like query 8, the speedup is as large as 28%.

Dynamic vs Static Lineage

If we could achieve good performance with static task dependencies determined before query execution, we would not need to log the lineage during query execution. In a *static lineage* strategy, a task consumes a fixed number of input data partitions at a time. If this number is too small, its outputs will also be smaller and there will be more tasks required for each channel, resulting in a higher volume of smaller partitions will be transmitted across the network, diminishing network efficiency. However, if this number is too big, effective pipelining cannot occur, and the system effectively executes in a stage-wise fashion similar to SparkSQL. It is very difficult to statically choose this number correctly in practice, since the sizes of data partitions can depend on the size of the cluster, data distributions and join and filter selectivity.

To demonstrate this difficulty, we show the performance of two static lineage strategies across the selected queries on the 4-worker and 16-worker clusters in Figure 2.12. In the first strategy, stateful operators batch together 8 input data partitions for each execution. In the second strategy, they batch together 128.

Similar to the previous experiment, performance differences between these strategies are not apparent for the simple queries in category I. However, differences become more significant with join queries in category II and more so with more complex joins in category III. We see that on the 4-worker cluster, a batch size of 8 is clearly superior to a batch size of 128, while the reverse is true on the 16-worker cluster.

On the 16-worker cluster, a batch size of 8 causes very small partitions to flow through the system, causing a marked decrease in CPU utilization and network I/O efficiency during shuffles. On the 4-worker cluster, the partition slices shuffled were larger due to the reduced parallelism, causing batch size 8 to instead outperform batch size 128.

Using dynamic task dependencies allows Quokka to achieve similar or better performance than the better of the two static lineage strategies in both cluster settings for most queries.

2.6.3 Write-ahead Lineage Overhead

In this section, we benchmark the overhead imposed by Quokka's write-ahead lineage algorithm during normal execution and compare it to spooling based options. We turn Trino's fault tolerance off to measure the overhead added by its HDFS spooling in Figure 2.13.[11]

Across the selected queries, Trino's spooling adds a geometric mean 1.5x overhead on the 4-worker cluster and 2.7x overhead on the 16-worker cluster, reaching up to 4.8x in the case of query 9. The overhead is considerably worse on the 16-machine cluster compared to the four-machine cluster. We believe that as the data partitions that need to be spooled to HDFS become smaller, HDFS efficiency markedly decreases. We also experimented with S3-based spooling for Trino, which led to much worse results.

We also implemented S3-based spooling in Quokka and observed similar overhead to Trino, as shown in Figure 2.13. Note Quokka's spooling overhead is minimal for the two queries in category I since Quokka's aggregation pushdown makes the spooled data size insignificant. It appears Trino does not perform this optimization. Quokka's spooling overheads are similar for simple joins in category II and complex joins in category III since most of the spooled data comes from the lineitems table, referenced by all the queries.

In comparison, the overhead of Quokka's write-ahead lineage strategy is an order of magnitude better than the spooling options, only 15% on the 4-worker cluster and 6% on the 16-worker cluster. Like Spark, Quokka backs up partitions unreliably in the worker's local disk instead of RAM to save

[11]The overhead is defined by the ratio of ratio of runtimes with and without fault tolerance. A value of 1 means there is no overhead.

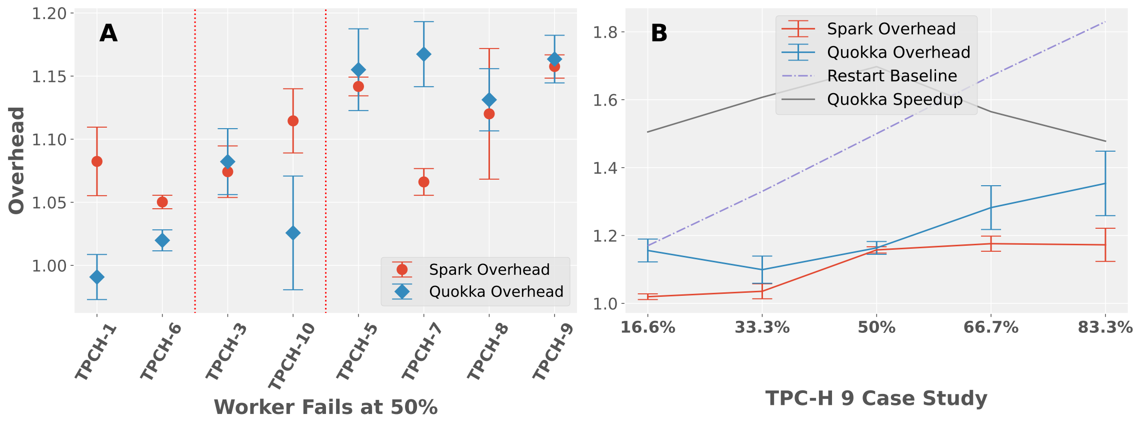


Figure 2.14: Quokka vs SparkSQL's fault recovery behavior. a) Quokka vs Spark fault recovery performance on the 16-worker cluster where a random worker is killed at 50% query completion during each query. b) A case study for TPC-H 9 where worker dies at varying points during the execution. We also show Quokka's end-to-end speedup over Spark on the same y-axis scale.

memory. However these local disk writes are a lot more efficient than networked HDFS or S3 writes, and can typically be hidden by computation and network IO.

Compared to Spark, Quokka also needs to consistently log the lineage of each spilled partition, which currently happens via the Redis GCS on the head node. We find this cost to be negligible in our benchmark as optimizations described in Section 2.4.1 greatly simplified the lineage. Virtually all the overhead results from the disk writes.

In addition to spooling, we also benchmarked Quokka with custom checkpointing strategies to S3. Even with incremental checkpointing, we observe severe overhead in normal operation. The biggest overheads come from operators whose state increases over time, like building the hash table for a shuffle hash join. While the exact overhead depends on the checkpointing interval, any reasonable interval that is useful for recovery performs much worse than spooling to S3.

2.6.4 Fault Recovery Performance

We now compare Quokka's fault recovery performance compared to SparkSQL. Instead of Spark-SQL's data parallel recovery, Quokka engages in pipelined parallel recovery as described in Section 2.4.2.

The first fault recovery experiment consists of running each of our representative queries on the 16-worker cluster. A worker machine is killed halfway through the query based on its normal execution runtime. The fault recovery overhead, defined by total runtime with failure divided by normal runtime without failure, is shown in Figure 2.14a for both systems.

We observe that Quokka and SparkSQL have similar recovery overhead, with Quokka's overhead better by a geometric mean of 1%. We see that for both SparkSQL and Quokka, simpler queries in Category I tend to have lower recovery overhead compared to more complicated joins in Categories II and III. While Quokka is faster than SparkSQL at fault recovery in Category I, it is slightly slower than SparkSQL in Category III. Note that in every case, we significantly outperform the baseline of just restarting the query from scratch on the remaining workers, which corresponds to an overhead of 1.5x.

In Figure 2.14b, we show a case study on TPC-H query 9 where we show SparkSQL and Quokka's fault recovery performance when the query experiences a failure at different points throughout the query. As expected, Quokka incurs higher fault recovery overhead if the failure occurs late in the query, as there is more work to be redone. SparkSQL exhibits the same behavior. However in all cases, Quokka and SparkSQL's fault recovery performances significantly beat the simple baseline of restarting the query from scratch after failure. Even though Quokka has more recovery overhead, it still outperforms Spark end-to-end with the failure in all cases since it is much faster in normal execution.

We notice that despite SparkSQL's usual good fault recovery performance, it occasionally fails to recover by continuing to make RPC requests to the dead worker, which is a known problem for open-source Spark on AWS EMR [6]. Our results only included trials where this problem is not encountered, which strictly improved SparkSQL's fault recovery results. Quokka's choice to communicate only through the GCS to avoid all direct RPCs between different components preclude it from this class of problems.

2.6.5 Scalability

Performance

We test the scalability of Quokka and write-ahead lineage with 32 rd6id.xlarge workers on the TPC-H benchmark queries with the same dataset. Figure 2.15a shows the speedup Quokka achieves vs. Spark and Trino in this setting. The speedup profile across the queries is largely similar to the 4-worker and 16-worker settings. Quokka achieves geomean 1.92x speedup over Spark and 1.86x over Trino. Quokka's speedup over SparkSQL is stable while its speedup over Trino improves with the number of machines, confirming our observations in Section VIC that Trino's spooling overhead gets worse with the number of machines.

Fault Recovery

Figure 2.15b repeats the experiment shown in Figure 2.14a, where a worker machine is killed 50% of the way through a representative query, for the 32-worker setting. We see that compared to the 16-worker setting, the recovery performance of Quokka deteriorates compared to Spark. On average, Quokka has 12% worse geomean recovery overhead in this case, vs. 1% better in the 16-worker

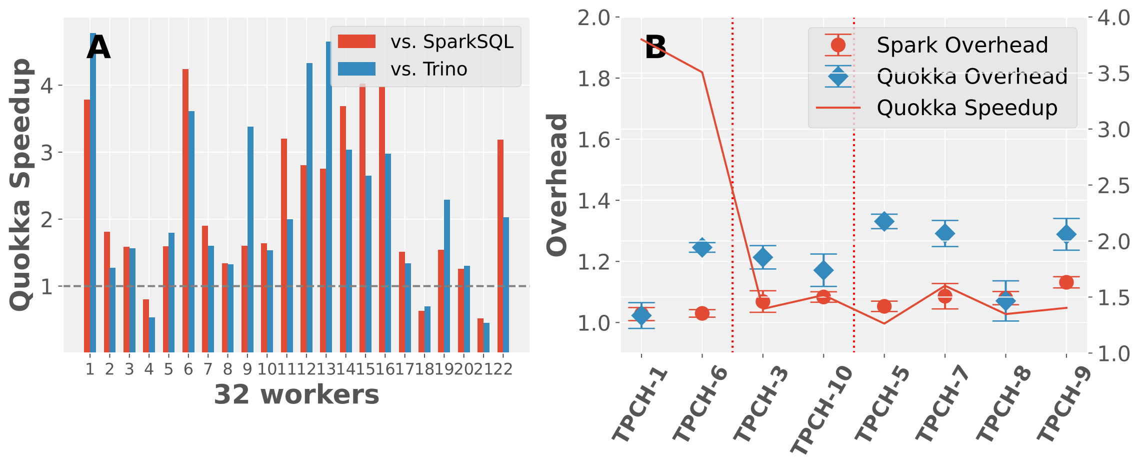


Figure 2.15: Experiments on 32 workers in terms of a) normal execution performance without failures compared to Spark and Trino with FT (TPC-H query number on x-axis) and b) fault recovery overheads where a random worker is killed at 50% completion. Quokka is still faster end-to-end on each query compared to SparkSQL (right y-axis, solid red line).

setting. However, even though Quokka has more recovery overhead, it both outperforms the restart baseline (1.5x overhead) in all cases and outperforms Spark end-to-end with the failure in all cases due to its faster normal execution performance.

Quokka's degraded fault recovery performance at 32 nodes can be understood from the discussion in Section 2.4.2. Unlike Spark's data parallel recovery, pipelined parallel recovery only leverages parallelism up to the number of stages in the query, not the number of workers in the cluster. As a result, increasing the worker count from 16 to 32 improves Spark's recovery performance but not Quokka's. Our design point exploits the fact that it is rare to see more than 16 nodes in practice, even for very large workloads [138].

2.6.6 Thinking about Mean Time Between Failures

We establish that Quokka has 1.92x speedup over Spark on 32 workers, with 12% worse geomean recovery overhead. If we assume that each failure introduces the same amount of fault recovery overhead, then Quokka and Spark's performance would breakeven when there are 5 failures given 32 workers. If we assume that Quokka's speedup and recovery overhead is stable across data scales and thus total runtime, then we can reason about the breakeven point of Quokka vs. Spark based on the relationship between the total runtime (X) and the mean time between failures (MTBF):

- Total system failure rate $= 32\lambda$ where λ is individual worker failure rate
- Set expected failures: $5 = 32\lambda \cdot X$

- Solve for failure rate: $\lambda = \frac{5}{32X}$
- Convert to MTBF: MTBF $= \frac{1}{\lambda} = \frac{32X}{5} = 6.4X$

This means for cases when the MTBF is less than 6.4 times the expected runtime of the job, Spark is expected to be more performant than Quokka end-to-end. While the total runtime can evidently change, so can the MTBF, based on the priority of the job, congestion on shared compute resources etc.

2.7 Discussion

We present write-ahead lineage, a novel fault tolerance mechanism for pipelined query engines with low overhead and fast recovery times. We showcase its implementation in a real query engine, Quokka, achieving 1.9x speedup against SparkSQL and 1.7x against Trino in normal operation on a 16-machine cluster on TPC-H, while matching SparkSQL in fault recovery performance. On 32 machines, Quokka maintains its strong performance while degrading slightly in fault recovery performance due to the reasons explained in Section 2.6.5. Ablation studies shown in Figure 7, 8 and 9 confirm Quokka's dynamic pipelined execution brings concrete performance benefits while write-ahead lineage incurs an order of magnitude less overhead compared to spooling-based alternatives.

2.7.1 Data vs Compute Fault Tolerance

This chapter talks a lot about fault tolerance in pipelined query engines. But why is it needed?

Pipelined query engines were originally designed to be part of a database server. When a machine failed, the first concern was preventing data loss and minimizing system downtime, not recovering a transient user query [86]. Intra-query fault tolerance became relevant in the era of decoupled compute and storage. Cloud storage now typically guarantees many nines of data durability, resolving concerns of data loss. On the other hand, computation is increasingly conducted by ephemeral resources that might fail or be pre-empted at any time.

In this setting, it makes sense to study how to recover from computation failures while assuming the input data is persistent and replayable, leading to popular fault-tolerant systems such as MapReduce and Spark [51, 152]. However, their stagewise design leads to inefficiencies in query execution. Pipelined query engines such as Trino have attempted to add fault tolerance, though its spooling based approach has high overhead [7]. Hosted query engines such as Snowflake are also fault tolerant to the user, but they simply restart failed queries under the hood [3].

2.7.2 Design Motivations and Novelty

Fault tolerance is a well-studied field with multiple established techniques such as lineage, spooling and checkpointing. While pipelined systems like Trino or StreamScope typically adopt a combination of spooling and checkpointing to achieve fault tolerance, we find these techniques cause high overheads in normal execution, as shown in Figure 2.13 [7, 82, 93].

Quokka's write-ahead lineage combines persistent lineage logging with upstream backup, similar to Spark. However, important differences exist due to Quokka's novel setting of pipelined query execution with dynamic task dependencies. In contrast to Spark's static lineage, write-ahead lineage logs the lineage as it is determined during query execution. We show this imposes negligible overhead in normal execution in Figure 2.13. Instead of Spark's data parallel recovery, Quokka adopts pipeline parallel recovery, whose degree of parallelism scales with the number of stages in the pipeline instead of the number of workers. We show that Quokka has comparable fault recovery overhead to Spark on the most common workloads with up to 16 workers, only tailing off at 32 nodes while still maintaining an end-to-end performance improvement because of the benefits of pipelined execution.

2.7.3 Pre-emption Warnings and Fault Tolerance

While Kubernetes pods can be killed without much warning, on popular public clouds such as AWS and GCP today, preemption notices ranging from 30 seconds to 2 minutes are sent out before a spot instance is reclaimed [20, 66]. Quokka does not currently account for such preemption warnings. Accounting for such warnings hold the promise of drastically speeding up fault recovery.

A simple approach could work as follows:

- When a TaskManager is notified that the worker node it is on will shut down soon, it will look at its assigned channels and the latest executable in each. The coordinator, also aware of this information, will dispatch to the TaskManager where each of those channels are rescheduled.
- The affected TaskManager will then stop making forward progress and instead try to use its remaining time alive sending all the state variables for these channels to the new TaskManagers. It should prioritize using its network bandwidth to send "complete" state snapshots, as partial transmissions are useless.
- If the new TaskManager received a complete copy of the state variable right before the current task of the affected channel, then the channel does not have to restart from the beginning, minimizing the amount of recomputation and data replays upon fault recovery.

2.7.4 Implementation

We believe simplicity is the most important part of distributed system design. Quokka's implementation is inspired by the design philosophy of outsourcing the control plane to an external data store [40, 82, 103, 127]. This design choice greatly simplified the core runtime implementation.

We have open sourced Quokka on Github[12] to facilitate data systems research and applications. Quokka supports a DataFrame API similar to Spark and Polars, and has already been used to support emerging data engineering applications like vector search on data lakes [8]. We believe write-ahead lineage can easily be added to any distributed pipelined query engine, where a distributed key-value store like DynamoDB or FoundationDB can be used to track metadata [48, 126, 156].

[12]https://github.com/marsupialtail/quokka

Chapter 3

Improving Processing: Cluster Heterogeneity

3.1 Motivations

In the previous chapter, we demonstrate a new fault tolerance algorithm for pipelined query engines to take advantage of the elasticity of the cloud. In this chapter, we examine how to leverage heterogeneity – where the compute resources used to execute a query could be composed of many different instance types.

While a large volume of work has focused on optimizing distributed query execution on a fixed cluster configuration, we focus on the emerging problem of ***optimizing the cluster configuration for a particular query***. This issue is gaining relevance as the fast VM spin-up time on public clouds has made it practical to spin up a dedicated cluster for each query instead of sharing a fixed cluster among queries, which enables highly desirable performance and security isolation amongst different data processing jobs. While already viable for long-running queries prevalent in data pipelines, new developments in micro-VMs promise to unlock this potential even for interactive queries [11].

The performance and cost-efficiency of a distributed query engine on any particular query could vary drastically depending on the VM instance types on which it is executed. EMR, the hosted Hadoop service offering SparkSQL and Trino on AWS, supports more than 400 VM options. Instance types on cloud providers like AWS belong to different classes, some optimized for compute with a higher vCPU to memory ratio, while others are tailored for IO performance. Figure 3.1 displays the computed cost per GB of memory and cost per Gbps of network bandwidth of different instance types on AWS. There can be a 14.7x difference in dollar cost per GB of RAM and a 61.7x (!) difference in dollar cost per Gbps of bandwidth between instances. Even among the much smaller set of instance types typically used for Spark or Trino clusters in practice, we commonly observe 3x

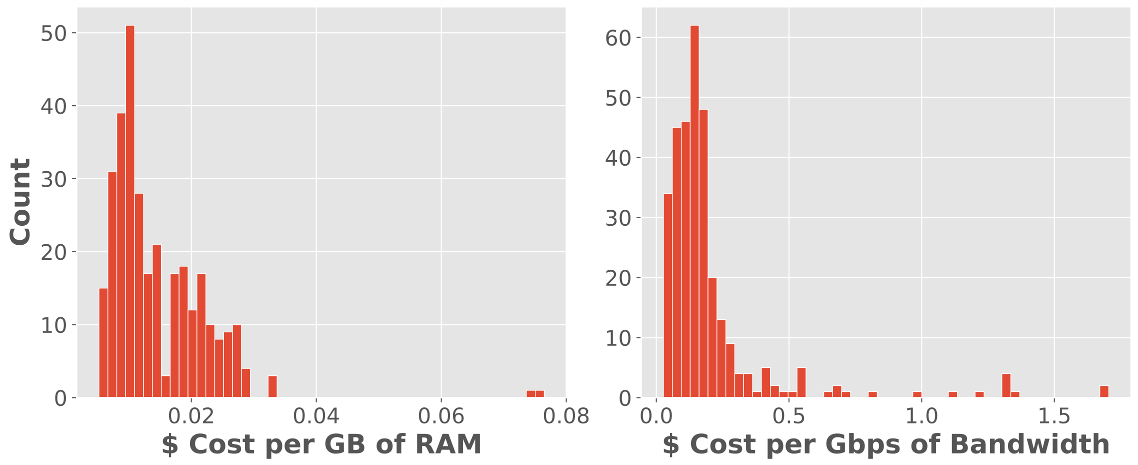


Figure 3.1: Histograms of memory and bandwidth costs of available instance types on AWS.

differences.

To the best of our knowledge, all current work on optimizing the cluster configuration for query processing has focused on *homogeneous clusters*, where the distributed query engine executes the entire query on a cluster consisting of a single instance type [15, 37, 73, 90, 150]. These systems vary in terms of the approach adopted to determine the optimal configuration (e.g. using Bayesian Optimization [15,73]), the execution of this approach (i.e., online vs. offline modeling), and the type of data integrated into the model (e.g. coarse-grained instance-level information, low-level metrics, etc.). These frameworks, however, ultimately consider only clusters of one specific instance type. Of particular interest is recent work by Leis and Kuschewski that proposes a mental model for optimizing instance type selection based on the Pareto optimal frontier of the query's total cost and the total execution time [90].

While an important first step, we believe homogeneous clusters do not fully exploit the resource flexibility offered on the public cloud today. This chapter studies the optimal cluster configuration problem for *heterogeneous clusters*, which might contain more than one instance type. We make the following contributions:

- We propose a Pareto optimal framework for evaluating cluster configurations in the context of heterogeneous clusters based on iso-cost curves, instead of the iso-instance curves proposed in [90].
- We apply the framework to left-deep join trees in pipelined query engines and provide intuition on why heterogeneous clusters are beneficial for cost efficiency.
- We evaluate heterogeneous clusters in a real pipelined query engine, showing performance benefits over homogeneous clusters on an example join.

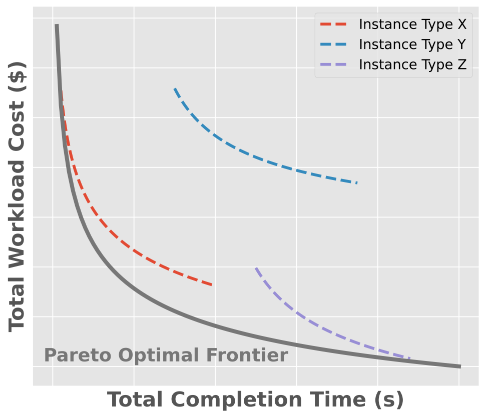


Figure 3.2: The model in [90] based on iso-instance curves.

This chapter is more about raising questions than answering them. We hope the reader will see that the discussion that ensues barely scratches the surface of this interesting topic, and naturally leads to many promising future research directions.

3.2 What is Optimal?

In this section, we propose a method to evaluate the optimality of a cluster configuration given a fixed query. Let us assume we have a distributed query engine and a query we would like to execute on some fixed input data. The distributed engine could be run on a *homogeneous* cluster, consisting of one instance type, or a *heterogeneous* cluster, with multiple different instance types.

The key question is: *How do we know the cluster configuration we picked is the best one?*

3.2.1 Pareto Optimality

To answer this question, we expand on the framework developed by Leis and Kuschewski [90], which only considered homogeneous clusters. The framework considers a graph with the x-axis representing the total execution time of the query and the y-axis representing the workload cost, as shown in Figure 3.2. Each point on the plot corresponds to a cluster configuration: if we assume that, given a cluster configuration, there is one way for the query engine to execute the query, then each configuration has exactly one fixed total completion time and workload cost.

The *Pareto optimal frontier* represents the lowest total workload cost for a fixed completion time. We should aim to pick cluster configurations that lie on this frontier. The Pareto frontier is parameterized by the query engine, the input data, and the query. Enhancing the engine's efficiency

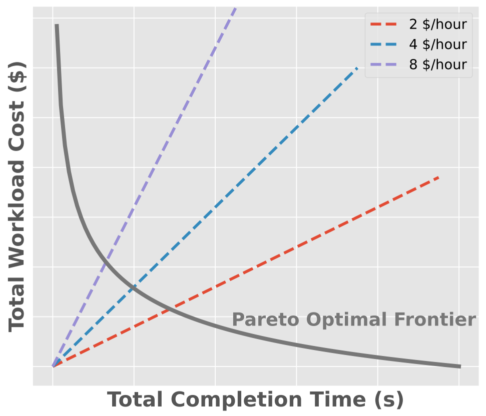


Figure 3.3: Our proposed framework based on iso-cost curves.

or simplifying the query shifts the curve to the left.

Leis and Kuschewski propose a model that predicts how the total workload cost and completion time vary as a function of the number of VMs in a homogeneous cluster for a specific instance type. This approach results in an *iso-instance* curve in the graph for each instance type. An iso-instance curve could intersect the Pareto frontier at more than one point, as shown in Figure 3.2.

3.2.2 Iso-instance vs. Iso-cost Curves

If we consider heterogeneous clusters, there are a very large number of such curves to draw since the potential cluster configurations grow exponentially with the number of different instance types considered. An alternative approach is to consider cluster configurations with the same cost per hour instead of the same instance type. Cluster configurations with the same unit-cost can be described by a straight line from the origin as shown in Figure 3.3, with the slope equal to the cost per hour of the cluster configurations on that line. We will refer to these lines as **iso-cost curves** (alternatively iso-cost lines). Importantly, iso-cost refers to the same cost per unit time. As one moves along the iso-cost curve, the total cost changes!

An iso-cost curve by itself is not tied to a specific query and is purely based on available cluster configurations by the cloud provider and instance costs. Given the query, the runtimes of the cluster configurations described by the curve will map to a set of points on this curve. However, the Pareto frontier is tied to a specific query, and it intersects these iso-cost curves at different points for different queries. We will show some real examples of iso-cost curves in Section 3.4.

An important benefit of thinking about cluster configurations in terms of these *iso-cost* curves is that, unlike iso-instance curves, each iso-cost curve intersects the Pareto frontier at exactly one point. In other words, among all possible cluster configurations with a particular cost per hour, there is only one cluster configuration that is optimal for a particular query.

In a heterogeneous cluster, a single cluster configuration could map to multiple points along the iso-cost curve if the query engine explores different mapping strategies of the query onto the processors, i.e. using different instance types for different stages[1]. Therefore, the optimal point on an iso-cost curve corresponds to the optimal mapping strategy of the optimal cluster configuration.

We can thus approach our key question using constrained optimization: **Among cluster configurations with the same cost per hour, which is the most efficient for the given query engine and query?**

3.2.3 Heterogeneity = More Choice

We hypothesize that the optimal point along an iso-cost curve is more likely to correspond to a heterogeneous cluster than a homogeneous cluster. To see why, let us introduce some formalism to describe a cluster configuration.

We assume our cluster is composed of instances from a set of N instance types, described by length-N vectors c, cpu, mem, and io. These vectors represent the per-hour cost, number of vCPUs, available RAM, and network bandwidth of each instance type. The cluster configuration can then be described by another vector n denoting how many of each instance type is in the cluster.

If we assume that the runtime for the optimal mapping strategy of a cluster configuration is a function H of the cluster's total resources, then we arrive at the following constrained optimization problem for the optimal configuration along an iso-cost curve:

$$\begin{aligned} \underset{\mathbf{n}}{\text{minimize}} \quad & H(\mathbf{cpu} \cdot \mathbf{n}, \mathbf{io} \cdot \mathbf{n}, \mathbf{mem} \cdot \mathbf{n}) \\ \text{subject to} \quad & \mathbf{c} \cdot \mathbf{n} = C \end{aligned} \tag{3.1}$$

We define new variables for total resources, $cpu = cpu \cdot n$, and similarly for mem and io. If we also define the set of all possible combinations of these values given the total cost constraint to be P, then we can rewrite Equation 3.1 in terms of total resources:

$$\begin{aligned} \underset{cpu, mem, io}{\text{minimize}} \quad & H(cpu, mem, io) \\ \text{subject to} \quad & (cpu, mem, io) \in P \end{aligned} \tag{3.2}$$

The problem boils down to optimizing a potentially hard-to-evaluate function H over a constraint set P, as shown in Figure 3.4. We refer to the constraint set as the **resource constraint polytope**. The benefit of using heterogeneous clusters lies in greatly expanding the size of this polytope by leveraging the vastly different resource-per-unit-cost characteristics of cloud instances. Note as

[1]This observation applies to homogeneous clusters as well, though it is not considered in [90].

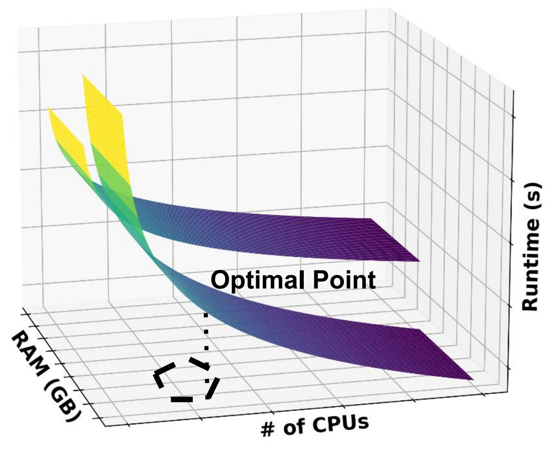


Figure 3.4: Optimizing H over the constraint polytope. For simplicity, the IO axis is ignored in the plot.

shown in Figure 3.4, H might not be continuous, but should be generally decreasing with increasing resources.

This model can be extended to incorporate resource types other than vCPU, IO, and total memory to account for different vCPU types like Gravitron or AMD. The constraint polytope would remain linear if the amount of this new resource increases linearly with the number of instances.

Figure 3.5 shows the significant resource flexibility achieved through heterogeneous instances, specifically in terms of total vCPU and RAM. The different colors represent constraint polytopes corresponding to different total cluster costs. For instance, the purple region in Figure 3.5 represents all possible combinations of total vCPU and RAM achievable for both (a) heterogeneous clusters and (b) homogeneous clusters with a total cost of \$4/hour, using three different instance types. Homogeneous clusters provide only three choices, where the maximum number of instances available for each type is selected within the cost constraint. In contrast, heterogeneous cluster configurations "connect" the vertices identified by the homogeneous clusters with the same cost.

Importantly, instead of allowing the users to obtain more of a particular resource type like total vCPUs for a particular price, heterogeneity offers more fine-grained *flexibility* in terms of trade-offs between different resource types.

3.3 Why is More Choice Beneficial?

Increasing the space of potential total resource combinations does not guarantee better query performance. For example, if the query engine is entirely CPU-bound, the optimal cluster configuration

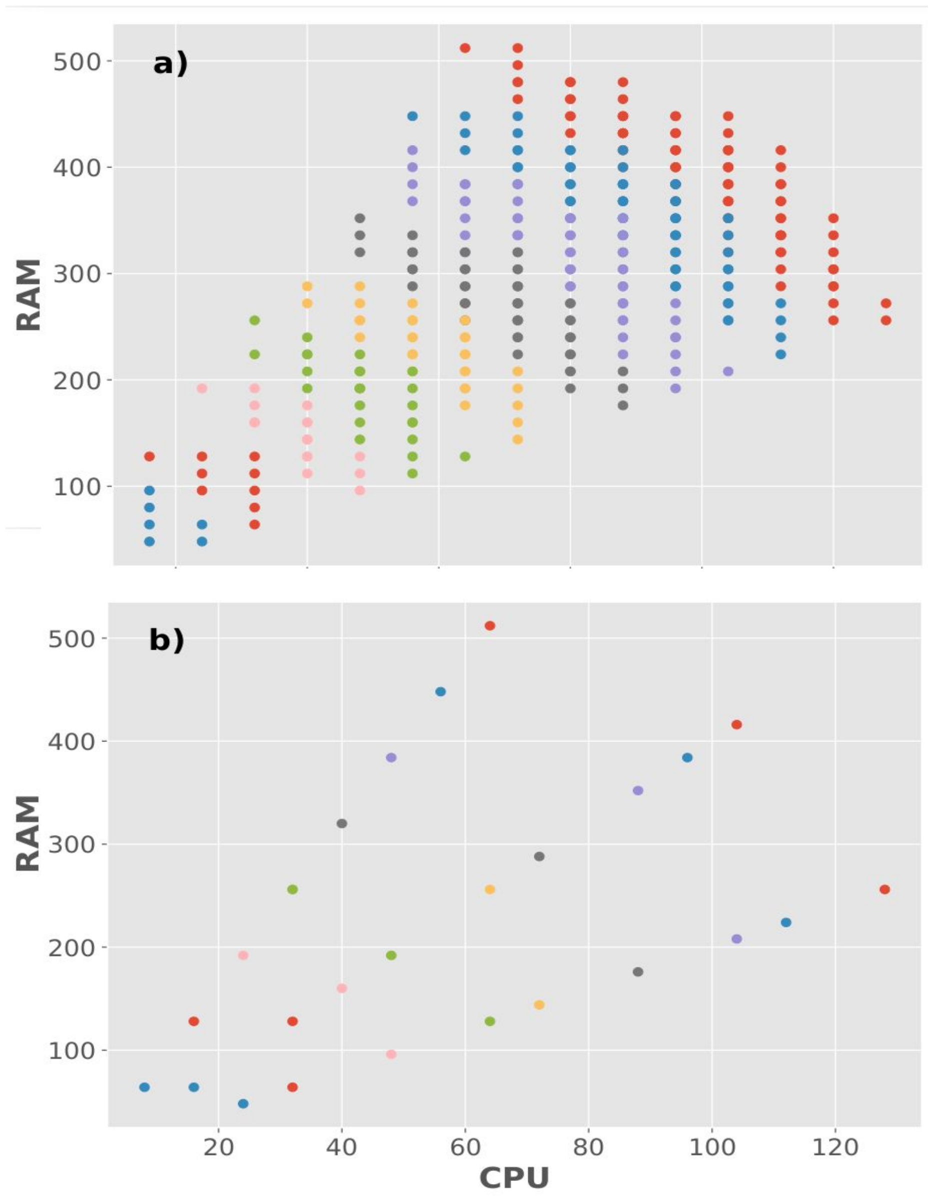


Figure 3.5: Available total resource combinations for fixed total cost for a) heterogeneous clusters and b) homogeneous clusters. Different colors correspond to different total cost per hour from \$.5 to \$5/hour in increments of \$0.5 from lower left to upper right.

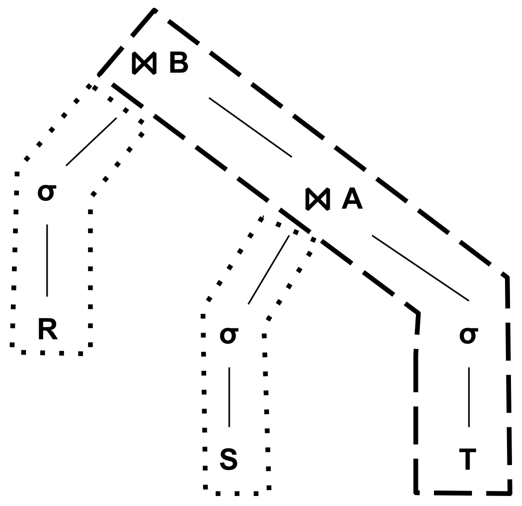


Figure 3.6: A left deep join tree. The query engine builds hash tables R and S (build side), and table T is probed against A and B in a pipeline to perform the three way join between tables R, S and T.

should consist solely of the instance type with the lowest cost per vCPU per hour.[2]

As illustrated in Figure 3.5, if the best performance is achieved by maximizing one resource type, homogeneous clusters would suffice. However, real query engines typically exhibit different resource demands when executing different parts of a query, which makes the increased flexibility in navigating the total resource space beneficial.

3.3.1 Query and Query Engine

To illustrate the potential benefits of this flexibility, we focus on the query execution of multi-stage joins in pipelined query engines, such as Snowflake, SingleStore, Trino, and DuckDB [43,48,89,124]. These queries are ubiquitous in production data pipelines.

The query planner typically executes a multi-stage join with a left-deep join tree as shown in Figure 3.6, which executes a series of distributed hash joins in a pipeline. Typically, the largest table is selected as a probe table, and the other tables serve as build tables. The join is executed in two phases. In the build phase, build tables R and S are read in parallel and hashed in memory with possible disk spilling. In the probe phase, probe table T is read and joined against the pre-built hash tables in a pipeline.

In a typical star-schema data warehouse, the build tables are a lot smaller than the probe table, which leads to the probe phase being the bottleneck. In this work, we will focus on the

[2]In this chapter, let's assume different CPU types like Arm and x86 have roughly the same performance.

performance of the probe phase. Consequently, our key question becomes very specific: **among cluster configurations on a fixed iso-cost curve, which is the best for the probe phase pipeline in a multi-stage join?**

3.3.2 A Concrete Example

We have seen that different cluster configurations allow us to navigate within the resource constraint polytope to optimize H as given in Equation 2. However, up until this point we have avoided speculating on the shape of H. How do we relate the total amount of resources to the runtime of the query?

For simplicity, let's consider just a two-stage probe phase pipeline, for example, this SQL query on the TPC-H dataset, which joins the lineitem table against the orders table.

Listing 3.1: Example join query

```sql
SELECT sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_discount) as sum_disc,
        sum(l_tax) as sum_tax,
        max(l_shipdate) as max_shipdate,
        max(l_commitdate) as max_commitdate,
        max(l_receiptdate) as max_receiptdate,
        sum(o_totalprice) as sum_charge,
        max(o_orderdate) as max_orderdate
FROM lineitem, orders
WHERE l_shipmode in ('SHIP', 'MAIL')
  and l_orderkey = o_orderkey;
```

Assuming `lineitem` is the probe table, a pipelined query engine like Trino would execute the probe phase pipeline as a sequence of a scan stage followed by a join stage [89, 124]. For example, here the scan stage would read lineitem table and filter on l_shipmode, whereas the join stage will probe the results of the scan stage against a hash table built on the orders table.

Batches produced by tasks in the scan stage can (and should) be consumed immediately by tasks in the join stage to avoid materializing the intermediate results as much as possible. The two stages effectively proceed concurrently, sharing resources in the cluster.

To understand how the performance of these two stages varies with the amount of vCPU, memory, and IO assigned to each stage, we benchmark the performance of the scan stage and the join stage independently. We assign various combinations of vCPU, memory, and IO resources to each stage, using the query shown in Listing 3.1 on TPC-H SF-100 with input in Parquet format stored on AWS S3.

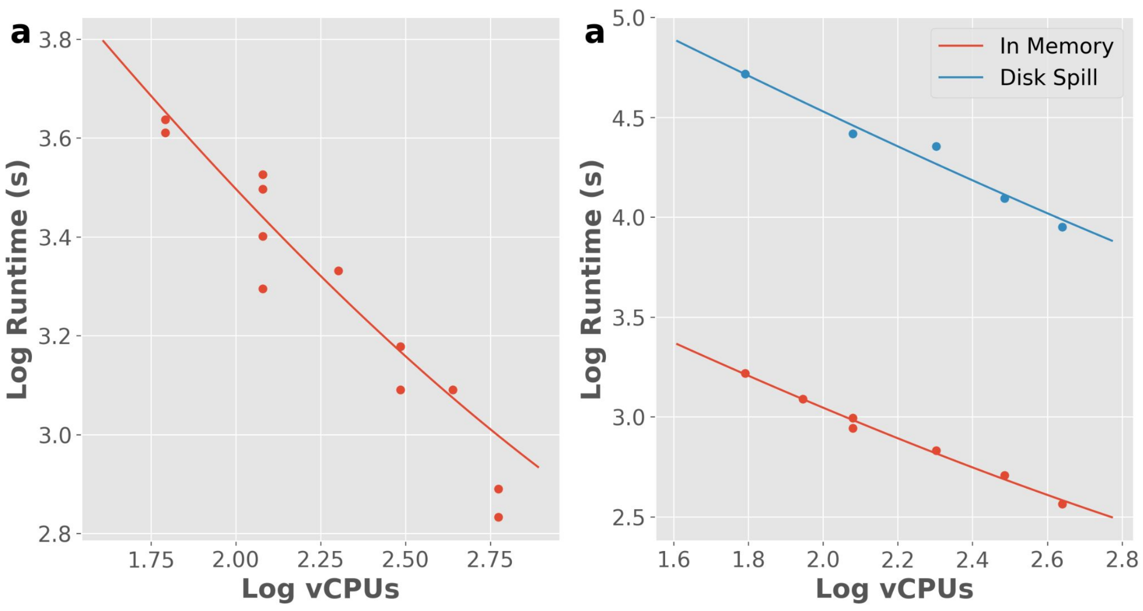


Figure 3.7: Performance for A) scan stage B) join stage as a function of assigned vCPU.

We use an open-source distributed query engine Quokka in our experiments [142]. Quokka's architecture resembles that of Trino, where tasks belonging to different stages in a pipeline are scheduled to executors on different machines by a centralized scheduler. On the TPC-H query benchmark, Quokka is able to achieve competitive results with Trino and SparkSQL. We select Quokka because it offers a simple interface to spin up heterogeneous clusters and it allows us to assign different resources to different query stages, a functionality missing in Trino or SparkSQL.

The benchmark results are shown in Figure 3.7. For the scan stage, we find that on the instance types we selected, the workload is entirely CPU-bound, as evidenced by a straight line fit the log-log plot of vCPUs against runtime. While one might expect network bandwidth to be the bottleneck, Quokka relies on open-source Parquet readers, which cannot saturate the high available network bandwidth per core. High-performance query engines like Clickhouse and DuckDB have developed their own highly optimized Parquet readers to mitigate this issue [45, 117], but other query engines like SparkSQL and Trino share this problem. In practice, we observe no performance difference when using network-optimized AWS instances and regular instances on TPC-H-like workloads on Trino, SparkSQL, or Quokka. In terms of memory, a fixed amount of memory is required by the tasks to parse Parquet files. Additional memory does not lead to a speedup.

The join stage's performance is dominated by both the available memory and the number of vCPUs. When there is not enough memory, Quokka has to spill the build side to disk, leading to substantial performance degradation. For both disk-based and in-memory joins, increasing the number of vCPUs results in increased performance. Like other open-source query engines, Quokka

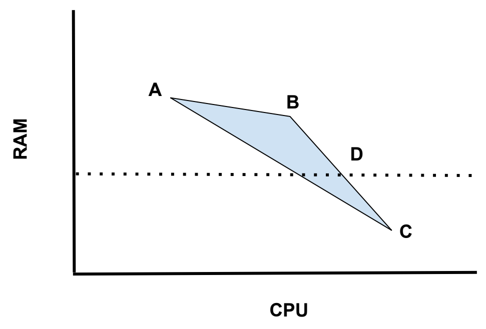


Figure 3.8: Illustration of the best cluster configurations achieved by heterogeneous clusters (D) vs homogeneous clusters (B). D has more CPU than B at the same cost per hour.

does not support graceful performance degradation.[3] Once the necessary amount of RAM is available to perform the in-memory join, additional RAM does not improve performance. Similar to the scan stage, we did not observe much impact on performance from the network bandwidth.

These results suggest that the shape of H resembles the surface in Figure 3.4: there is a qualitative shift in the shape of the function when there is enough RAM to perform an in-memory join. This analysis suggests a simple heuristic to select the optimal cluster configuration:

- If the largest total memory in the constraint polytope is less than what is needed for an in-memory join, the amount of total memory in the cluster becomes less relevant than the total number of vCPUs. We should just maximize the number of vCPUs in the cluster. Since the query is bottlenecked by one resource (vCPU), heterogeneous clusters will not help.

- If the polytope contains configurations with enough memory to perform an in-memory join, then use the configuration that has exactly this much memory and as many vCPUs as possible, since additional memory does not contribute to higher performance. Heterogeneous clusters are helpful in this case.[4]

Figure 3.8 illustrates why heterogeneous clusters are useful in the latter case. We have denoted the polytope in blue. As we have shown before in Figure 3.5, homogeneous clusters only hit the vertices. However, the most efficient total resource combination occurs on the line that connects vertices B and C, which can be approached more accurately with a heterogeneous cluster. To construct cluster configurations that connect B and C, one simply has to combine the instance types that would have made up the homogeneous clusters at B and C and mix the two in different ratios.

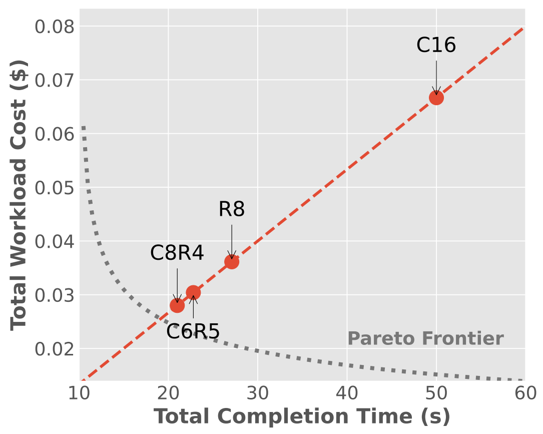


Figure 3.9: Speedups achieved by using heterogeneous clusters along the iso-cost curve with a unit cost of $4.8/hour. C8R4 denotes cluster configuration with eight c6gd.2xlarge and four r6id.2xlarge.

Instance Type (Abbreviation)	**vCPUs**	**RAM (GB)**	**$/hr**
r6id.2xlarge (R)	8	64	0.61
c6gd.2xlarge (C)	8	16	0.31
m6id.2xlarge (M)	8	32	0.48

Table 3.1: AWS EC2 instance specifications and pricing

3.4 Evaluation

We take the query shown in Listing 1 and explore whether using heterogeneous clusters can lead to real speedups in the pipelined probe phase. We consider heterogeneous clusters consisting of only two instance types, r6id.2xlarge and c6gd.2xlarge. We will consider another instance type in a more complex example to follow.[5] All instance types and their specs are shown in Table 3.1. The hourly on-demand pricing for the r6id.2xlarge instance is approximately double that of the c6gd.2xlarge instance. The two have the same number of vCPUs while the former has 4x the RAM per instance, which makes the former more cost-efficient for RAM and the latter more cost-efficient for vCPUs.

We consider a cost budget of using eight r6id.2xlarge instances or sixteen c6gd.2xlarge instances. We could trade off one r6id.2xlarge instance for two c6gd.2xlarge instances in our cluster while maintaining the same cost per hour, which in terms of total resources amounts to moving on the line between points B and C in Figure 3.8.

[3]Trino would typically run out of memory while SparkSQL defaults to a more expensive disk-based join.

[4]One would still need to be intelligent about instance selection to support mapping.

[5]We do not consider different sizes of each instance (i.e. 4xlarge, 8xlarge, etc.), as distributed database offerings typically rely on a fixed instance size. The 2xlarge setting (8 vCPUs) appears to be what Snowflake uses in practice [5] and what Databricks used for the TPC-DS benchmark [2].

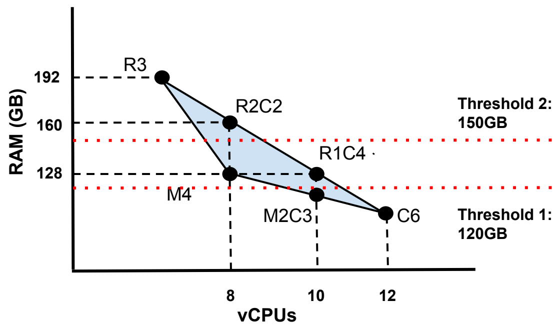


Figure 3.10: Illustration of the polytope depicted in Figure 3.8 for r6id.2xlarge, m6id.2xlarge and c6gd.2xlarge instance types with a total cost of around \$1.8 an hour on demand. Valid cluster configurations are shown by solid dots.

Figure 3.9 shows the concrete performance tradeoffs made as we explore heterogeneous cluster configurations that lie on that line. We start from eight r6id.2xlarge instances (R8)—offering the most RAM but the fewest CPUs—which allows us to do an in-memory join with a total completion time of 27s. As we use more c6gd.2xlarge instances in our cluster, we move to configurations such as C8R4 (eight c6gd.2xlarge and four r6id.2xlarge instances) and C6R5. These configurations have less RAM but still enough to execute the hash join purely in memory. However, these configurations have more CPUs, leading to a much faster total runtime of 20s and 21s, amounting to a 35% speedup over the R8 configuration. As we move purely to c6gd.2xlarge instances we have much less RAM and cannot execute the join in memory, which forces the query engine to use disk-spilling and severely degrades performance, leading to a runtime of 50s, more than two times slower than the C8R4 configuration.

The evaluation illustrated in Figure 3.9 uses the framework laid out in Section 3.2, where we move along an iso-cost curve. We see that the heterogeneous cluster configurations bring us closer to the hypothetical Pareto optimal frontier to the lower left. Since we are only exploring a very limited subspace of cluster configurations on this iso-cost curve, we cannot definitively state that any of our sampled configurations are "Pareto optimal". However, we see that the heterogeneous configurations C8R4 and C6R5 perform strictly better than the homogeneous configurations R8 and C16.

3.4.1 A More Complex Example

We now consider a more complex example with three different instance types, r6id.2xlarge, c6gd.2xlarge and m6id.2xlarge. For about \$1.8/hour, we can have a cluster of three r6id.2xlarge instances, or four m6id.2xlarge instances or six c6gd.2xlarge instances. We consider a more complicated join query involving three tables based on TPC-H 3, shown in Listing 2:

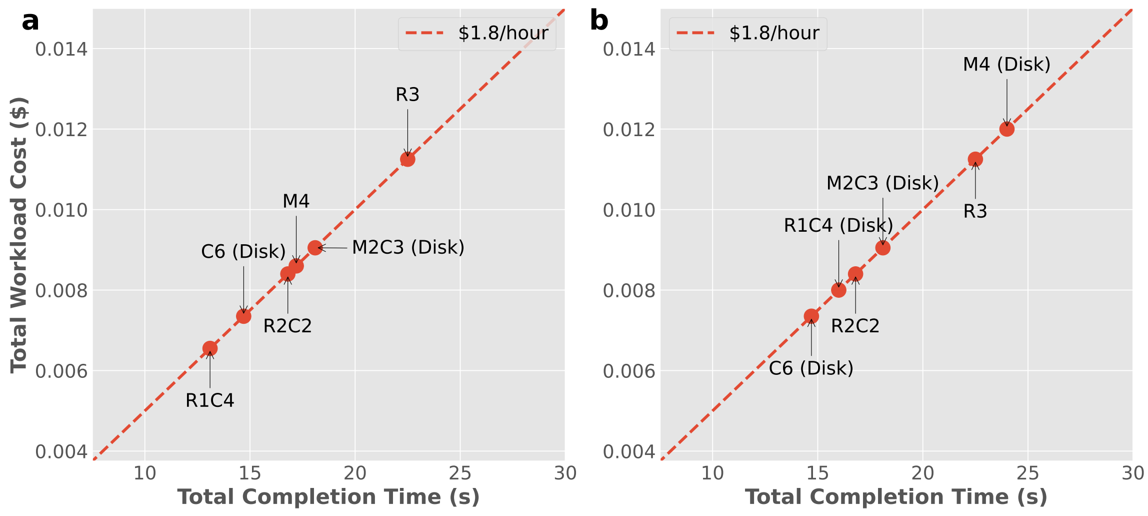


Figure 3.11: Runtimes of cluster configurations in Figure 3.10 assuming a disk-spilling threshold of a) 120GB and b) 150GB. The best performing cluster configuration is on the lower left.

Listing 3.2: Multi-stage join query

```
SELECT l_orderkey, o_orderdate, o_shippriority
        sum(l_extendedprice * (1 - l_discount))
            as revenue,
FROM customer, orders, lineitem
where c_custkey = o_custkey
      and l_orderkey = o_orderkey
      and o_orderdate < date '1995-03-15'
      and l_shipdate > date '1995-03-15'
group by l_orderkey, o_orderdate, o_shippriority
```

In Figure 3.10 we show what the constraint polytope shown abstractly in Figure 3.8 looks like for these three instance types. Besides the homogenous cluster configurations, we also consider three heterogeneous cluster configurations: R2C2 (two r6id.2xlarge instances and two c6gd.2xlarge instances), R1C4 and M2C3.

We consider two different benchmakrs with different RAM thresholds to force disk-spilling of the joins at 120GB and 150GB. The performance of each cluster configuration under these two settings is shown in Figure 3.11 along the iso-cost curve with slope $1.8/hour. We mark what cluster configurations are forced to disk-spill for the join in each case.

We briefly discuss how to interpret the results. For Figure 3.11a where the threshold is 120GB, we can see in Figure 3.10 that R1C4, the cluster configuration that performs the best, is directly above the cutoff, meaning it has the most vCPUs out of all cluster configurations that do not have to disk spill.

However, for a threshold of 150GB, the analogous cluster configuration R2C2 is outperformed by both R1C4 and C6, which use disk spilling. The query in Listing 2 is impacted by disk spilling less than the query in Listing 1, and the extra vCPUs overcome the benefits of avoiding disk spills. This example suggests that for some queries, there is a need for more accurate performance modeling than our simple heuristic to obtain the most CPUs without disk spilling.

Another observation is that cluster configurations involving the m6id.2xlarge (M) instance type are not among the top-performing configurations in either case—see Figure 3.10. Note that for any valid cluster configuration that includes an M instance, we can find a configuration consisting of only R and C instances by finding its horizontal or vertical projection onto the line connecting R3 and C6. Examples are shown for the cluster configuration M4 and M2C3, which suggests that the m6id.2xlarge instance type is not as cost effective as the r6id.2xlarge and c6gd.2xlarge instance types in terms of RAM or CPU.

3.5 Conclusion

In this chapter, we show that using different instance types in the same cluster can speed up query execution. We introduce iso-cost curves, which offer a simple method to reason about the optimality of a cluster configuration when optimizing query runtime with constraints on overall resources. We demonstrate that the key benefit of cluster heterogeneity is expanding the space of total resources available, leading to concrete performance benefits for the probe phase of a pipelined join.

This chapter is meant to raise questions, not answer them. We believe we have barely scratched the surface on the promising area of effectively leveraging cluster heterogeneity. For example, here are three promising directions to pursue for future work:

- **Different mapping strategies** While we focused on optimizing different cluster configurations, pipelined query engines support different ways to execute a query given a fixed set of instances. [6] Being able to quickly decide on the strategy given a query, a query engine, and a cluster configuration is far from trivial. Techniques employed in prior work such as Selecta and FineQuery could be applied to this problem [81, 139].

- **Model-guided cluster configuration optimization** This chapter presents a heuristic for a given query pattern based on performance benchmarking. It is impractical for practitioners to benchmark every query they encounter. A solution could be extensive offline benchmarking of different query execution stages coupled with a fast online cost-function-guided search for

[6] Assume we have a single join stage following a scan stage and two worker machines with two executor slots per machine. The engine could assign one executor slot per machine to each stage, the default strategy used in experiments here, or assign one machine completely to each stage.

a new query. We believe this approach, pioneered by FlexFlow for deep learning problems, holds great promise in making heterogeneous clusters practical for real query workloads [78].

- **Virtual clusters** In cases where a cluster manager like Kubernetes is used, the question then becomes how to properly configure the resource requirements of different pods used in a data processing job. The per unit resource cost of a pod is much harder to reason about than VM on-demand pricing, as it may involve business costs of preempting other jobs. Heterogeneity is still interesting, as e.g. using a mix of high and low-memory pods for a job might be easier to schedule than uniformly mid-memory pods.

Chapter 4

Improving Storage: Indexing Data Lakes

4.1 Motivations

Chapters 2 and 3 focus on improving distributed query processing. We now turn our attention to data storage. As discussed at the end of Chapter 1, data lakes have become the storage mechanism of choice for analytical data processing on the cloud today. Data lakes typically consist of Parquet files on object storage such as AWS S3 organized by a table format like Delta Lake, Iceberg or Hudi [24–28]. The Parquet files are queried on-demand with engines such as Trino, Databricks, Snowflake, or DuckDB [10, 29, 49, 108, 134]. These data lakes, or "lakehouses", have become the de facto centralized analytical data store for many organizations. However, they are still unsuitable for emerging workloads that are *search-oriented*. These workloads, such as high-cardinality time series, text, and embedding analytics typically need to quickly drill down to a very small subset of the data and perform complex aggregations on this subset. In most cases, the minimal indexing available in current Parquet-based data lakes does not allow efficient evaluation of the filter conditions (e.g. UUID match, text regex, approximate nearest neighbors), making these workloads inefficient for query engines like SparkSQL or Trino [30, 134]. As a result, the lakehouse paradigm breaks for these workloads – organizations have to duplicate their data to a specialized data management system like Clickhouse, ElasticSearch or Qdrant to perform efficient search analytics, as shown in Figure 4.1.

Some examples of search workloads:

- **High-cardinality filtering**, i.e. exact-matching on a column with an extremely high number of unique string values, such as UUIDs or sha256 hashes. Examples include observability workloads (filtering by Kubernetes pod name) and blockchain analytics (filtering by transaction hash).

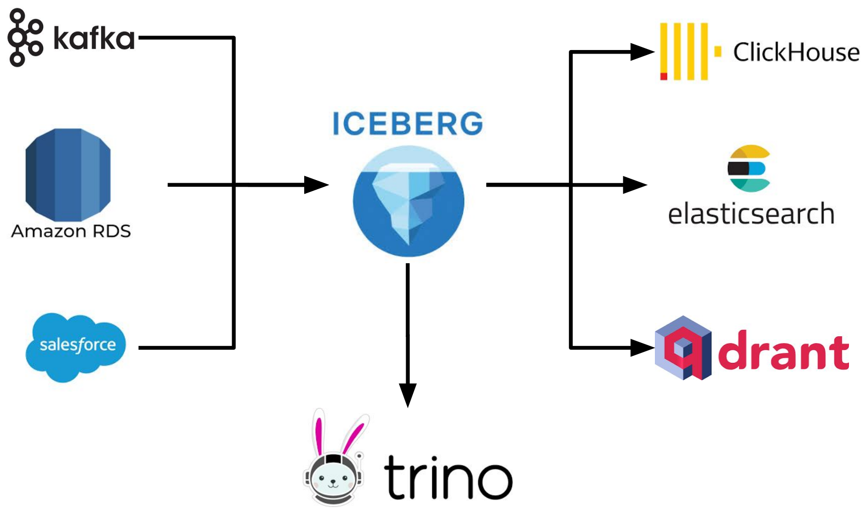


Figure 4.1: Typical enterprise data stack. For workloads where directly querying the data lake with a query engine like Trino is too inefficient, a specialized system like Clickhouse, ElasticSearch or Qdrant is used.

- **Substring search**, such as log search or text analytics. For example, to detect if evaluation datasets are leaked in the pretraining corpus, one could perform a substring search against the training records, which might be stored as a text column in a data lake.
- **Vector embedding search**, i.e. approximate nearest neighbor search. Examples include retrieval-augmented-generation (RAG) and recommendation systems.

Why do Parquet-based data lakes struggle on those workloads?

1. **Useless indices.** Parquet-based data lakes rely on sort orders to enable file skipping for good read performance by query engines. In search workloads, it is often infeasible to keep the data sorted on the column you want and maintain insertion performance. For example, time series data such as IT monitoring and blockchain logs are naturally sorted by timestamp, and sorting them based on UUID tags amount to an extremely expensive transpose. Natural sort-orders do not exist for text or vector embeddings. These two factors render min-max indices on column chunks useless for search workloads.
2. **Read granularity.** Even if an appropriate index exists, column chunks corresponding to text or binary data types typically tend to be tens to hundreds of MBs in size due to Parquet's design (further explored in Section 4.4.1). This means for highly selective search queries, current query engines have to retrieve many MBs of data from object storage just to return a result that is tens of bytes.

4.1.1 Existing Approaches

Today practitioners commonly employ two approaches to tackle such search workloads, ETL and brute force scan. We also mention an emerging third approach, which is new data formats.

Copy Data

Practitioners today commonly copy data from the data lake back into specialized systems tailor-made for their workload, such as Clickhouse [45], ElasticSearch [57], vector databases like Qdrant [115], or simply cache results of selective table scans [121] (Figure 4.1). While this approach guarantees the best performance for those queries by leveraging specialized systems, it largely negates the benefits of the data lake for the data in question. These systems tend to be compute-storage integrated and are expensive to manage. In addition, the data pipelines duplicate data and reintroduce data consistency and staleness issues that data lakes seek to address in the first place.

While we believe that this is the best approach if the search queries are frequent or require strict latency SLAs, e.g. a search engine like Google, we observe that many important workloads do not have such requirements. For example, an LLM pretraining data exploration system might serve only up to 100 internal users while query latencies up to a minute are acceptable. Similarly, a log management system might serve only a particular SRE team where query latencies up to a few seconds are acceptable.

Brute Force Scan

Another popular approach is to simply pay the high IO and compute cost of these queries by horizontally scaling the query engine and scan everything, such as with Spark [30]. Cloud vendors are especially incentivized to take this approach as they typically charge based on the amount of data scanned. For example, AWS Athena (based on Trino) has been heavily used by cybersecurity professionals to query logs, with a pricing model that reflects the brute-force scanning approach.

This approach is the simplest and most economical if querying is highly infrequent, since it incurs no upfront indexing compute or storage costs. However, it simply defers those costs to query time. While we believe this may be the best approach for workloads that might not ever read most of the data (e.g. log analytics), if most data will be read at least a couple of times over its lifetime, some form of indexing becomes attractive.[1]

New File Format

A third approach is to replace Parquet files with new file formats, such as Lance and Nimble [59,87]. While these new file formats are purposefully designed to excel at these new search-style workloads, they have not yet achieved the widespread ecosystem integrations that made Parquet appealing in

[1] Of course, this trade-off depends on the cost of computing and storing the index.

the first place. For example, all three major data lake formats only support Parquet today [24,25,28]. As a result, it is impractical today for most organizations to adopt them as the ground truth data store. This means in practice organizations still have to keep data in Parquet, making this no different than the copy-data approach.

Another similar approach is recent modifications to the Parquet file itself, for example column indices and bloom filters [32, 75]. While these approaches could be somewhat effective in some search queries that involve high cardinality tags, they do not help for text or vector searches. More importantly, they assume a Parquet writer that generates these additional indexing structures, which in practice is quite rare.

4.2 Rottnest Search Indices

Inspired by prior works which added transactional support to data lakes by adopting lightweight metadata alongside the Parquet files [24, 27, 62], we explore if we could support these novel search workloads through *additional lightweight index files on top of transactional data lakes*. We build Rottnest, a system that augments data lakes with external indices resident on object storage for workloads such as high cardinality analytics, full text search, and vector embedding search.

The transactional semantics of current data lakes are not meant to replace dedicated transactional databases in online workloads due to their high latency and low throughput. However, they are "good enough" for many offline workloads that require certain transactional semantics on big data. Similarly, Rottnest does not aim to replace specialized full-text or vector databases for online serving scenarios with high throughput and low latency SLAs. However, we believe it can be the most compelling option for most offline workloads such as historical log analytics and LLM pretraining data exploration.

Rottnest is designed to be the most economical solution for offline or human-interactive workloads with medium total query load, as shown in Figure 4.2. Similar to brute-force, Rottnest cannot support workloads below a certain latency threshold, typically a few hundred milliseconds, due to its need to access object storage. However, as we will show, this threshold is much lower for Rottnest compared to brute force.[2]

The design of Rottnest indices focuses on maximizing the purple-shaded area in Figure 4.2. On the left side, brute force requires no indexing cost but incurs very high search cost. The cheaper it is to construct a Rottnest index, the lower the total query load at which Rottnest pays back this cost through cheaper searches. On the right side, dedicated systems like ElasticSearch or vector databases require high upfront costs to store the index in SSDs or memory, yet search is fast and almost free. If we lower the cost of querying Rottnest indices, the relative search advantage of dedicated systems

[2]Recent technologies like S3 Express have theoretically lowered this latency threshold, though it remains unlikely for the Parquet data lake to be stored entirely in this tier due to cost.

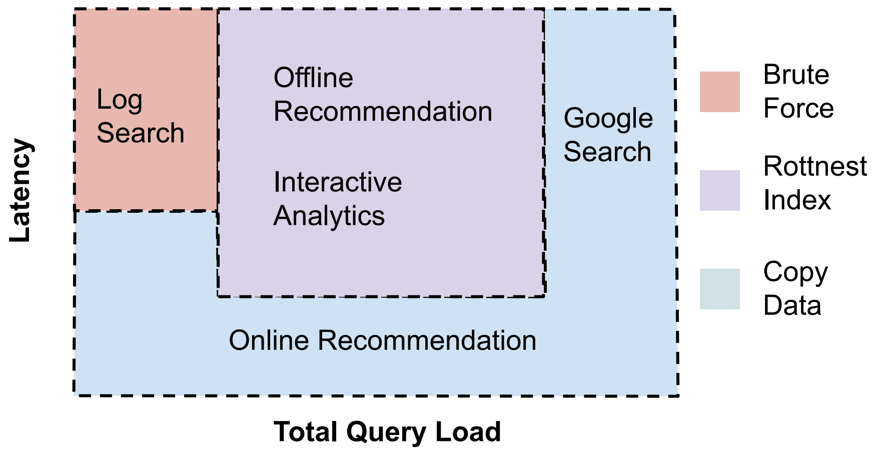


Figure 4.2: The most economical approach given latency and throughput requirements of the application. Some example applications are shown in each category.

decrease and Rottnest remains more economical at higher query load. This analysis leads us to our first two design goals:

- **Low indexing cost**, consisting of both the compute cost to construct and maintain the index, as well as the storage cost for the index files themselves.
- **Efficient querying** of the index files and relevant parts of the raw data.

Our objective of easy integration with existing data lakes to reduce data silos leads us to two more design goal:

- **On demand consistency** with the underlying data lake. Index construction should be completely decoupled from and invisible to current data ingestion and maintenance jobs. Rather than constructing the index when the data is ingested, the index should be constructed in a separate process at a time of the user's choosing. It is unacceptable to slow down or otherwise complicate existing write and read workloads. Ideally, the Rottnest index should be invisible to all writers and readers who do not need to use it. This marks a key difference between Rottnest and current database or data lake indices like Postgres indices and Hudi's upsert index, which slow down writes.
- **Generality**: Rottnest indices should be general and support a variety of different search query types. Ideally, we keep a single copy of the data, and simply "bolt-on" different indices.

4.2.1 Design Decisions

To achieve the design goals, we outline a sequence of core design choices of Rottnest. We justify how each design choice furthers one design goal at the expense of another.

Object-store Native Indices

Abiding by the design philosophy of data lakes, which put additional metadata structures on object stores, Rottnest indices are by design stored on object stores. Benefits of this design include compatibility, cost, and scalability as outlined in [27]. While this makes accessing the indices more expensive, Rottnest employs numerous optimizations to mitigate this, described in Section 4.4.2.

Eager vs. Lazy Indexing

Typically, indices in databases are updated upon data insertion or mutation. This “eager” indexing is natural when the same system, e.g. Postgres, is in charge of both maintaining the index and writing the data. However, Rottnest is aimed at speeding up certain classes of queries. It is not acceptable for its construction or maintenance to slow down data lake maintenance, data ingestion jobs, or other queries.

We opt for lazy indexing, where the Rottnest index is kept consistent with the underlying data lake on-demand, or periodically through a separate service like Airflow [23]. This means that the newest data to be queried will often be unindexed and has to be full-scanned similar to memtables in LSM-based data stores like RocksDB [53].

In-Situ Indexing

To lower the indexing overhead, the index files do not store a copy of the data, instead opting to query the data *in-situ* in the Parquet files, inspired by previous work such as NoDB [14]. Assuming the index structure is smaller than the compressed raw data, this drastically lowers the index storage overhead. We will show that useful indices can be constructed for search workloads that are much smaller than the compressed data. The techniques employed generalize over different workloads, e.g. substring, UUID and vector nearest neighbor search.

This design choice sacrifices query efficiency for low indexing overhead. Recent work has suggested that Parquet files have efficiency issues that limit their performance on highly selective search queries [83, 153]. However we will show most of these issues can be circumvented with a custom Parquet reader implementation described in Section 4.4.1, leading to search performance that rivals custom formats like Lance [87].

4.2.2 Summary and Implementation

To summarize, Rottnest constructs object-storage native index files that are kept lazily up to date with the underlying data lake. Each **index file** points a search query at physical locations in a collection of Parquet files. The index files are uploaded to an object storage bucket. Rottnest also keeps a **metadata table** that track the relationship between index files and the underlying Parquet files. The metadata table must support transactional updates, similar to the data lake we are trying to index. In fact, Rottnest defaults to use Apache Iceberg to implement this metadata table.

Searchers interact first with the metadata table, then reads Rottnest index files and Parquet files from the underlying data lake directly from object storage to complete the query.[3] Similar to data lakes, the Rottnest index files can be compacted and vacuumed.

Similar to delta-rs [119], the Rust-based client of Delta Lake, and pyiceberg, the Python client of Apache Iceberg, Rottnest is designed as an embedded library with index management APIs in Python. In the next section, we will explain Rottnest's index, search and compaction protocol in much more detail.

4.3 Index Protocol

The Rottnest client library supports four APIs: `index`, `search`, `compact` and `vacuum`. Of these, `index`, `compact`, and `vacuum` mutate the Rottnest index, maintaining two invariants that together guarantee correct `search`:

- **Existence**: all indexed files referenced in the metadata table are present in the object storage bucket.
- **Consistency**: an index file correctly indexes the associated Parquet files if they still exist.

We now discuss the four APIs.

4.3.1 Building a Rottnest Index

`index(table, column, index_dir, type):`

table: the data lake table to index.

column: the column in the data lake on which to build an index. Typically this is a string or binary column.

index_dir: the object storage bucket to store the index files. This can also be a local directory on disk for testing purposes.

[3]Rottnest by default does not assume or require any form of caching.

type: the type of index to build (BM25, keyword, high cardinality, ANN, etc.). Each type of index might also have associated keyword parameters.

Indexers call this API to keep the Rottnest index at `index_dir` up to date with the current *snapshot* of the data lake table. (Data lakes support time travel; a snapshot is a point-in-time copy of the data represented by a list of Parquet files in the snapshot.) The design would be simple for append-only data lakes where Parquet files can only be added: when `index` is called, build a new index file covering the new Parquet files. However, data lakes support operations that may invalidate existing Parquet files (e.g., compactions, vacuums, Z-order, and CRUD operations) as well as produce custom files such as *deletion vectors* which record individual rows of a table that have been deleted [100, 133]. Because our indices point to physical locations, index files may be invalidated by such operations.

We propose a simple protocol to ensure consistency in the face of these different data lake operations: index all new Parquet files written to the data lake, regardless of whether they result from insertions, compactions or updates. While searching, search only index files that include physical locations included in the snapshot. An index file might also map a query term to physical locations not in the snapshot – such locations are filtered out during the search.

To facilitate this process, Rottnest keeps track of the list of Parquet files it has already indexed in the Rottnest metadata table, which is implemented as a Delta Lake table itself resident on object storage. In principle, any transactional data store, e.g. Postgres, can be used for this purpose.

The indexing works as follows (example shown in Figure 4.3):

1. **Plan**: Look at the manifest list of the latest data lake snapshot and find the Parquet files that are in the current snapshot but not yet indexed. Rottnest only indexes new data files (d, e.parquet in Figure 4) and not deletion files (dv.bin).

2. **Index**: Rottnest proceeds to build an index file (ac02.index) that covers all the new Parquet files, and uploads it to `index_dir`.[4] If, for some reason, a file is no longer available to read during the indexing process, e.g. due to garbage collection of the data lake, the index API aborts and needs to be retried.

3. **Commit**: After the new index file has been uploaded to object storage, the indexer inserts a record into the Rottnest metadata table transactionally. Note that the metadata table shown in Figure 4.3 is simplified, in practice other metadata information such as total number of rows indexed and index timestamp can be recorded as well.

4. **Timeout**: If the index operation is not completed within a set timeout, it will abort and needs to be retried. The timeout is critical for garbage collection, as described later in `vacuum`.

[4]Some types of indexes (such as vector indexes) might have a minimum size limit. If the total number of rows in the new files falls below this limit, the indexing will be aborted in favor of brute force scan.

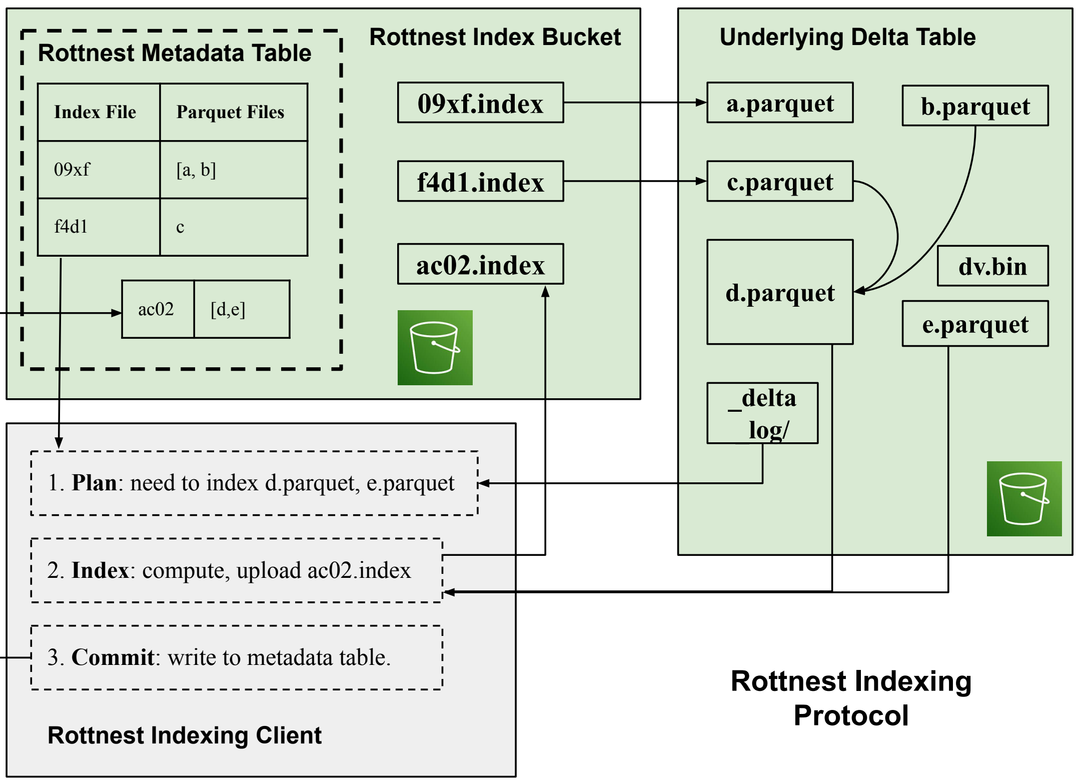


Figure 4.3: Rottnest Indexing Protocol. Since the last `index` call b.parquet and c.parquet has been compacted into d.parquet, and an update was written with the update file e.parquet and deletion vector dv.bin.

We do not require all index files present in the Rottnest index bucket to have an associated entry in the metadata table, which might occur if the indexer fails during commit. These index files will be garbage collected separately, described Section 4.3.3. We also do not require all Parquet files referenced by index files to still be active in the underlying data lake. Indeed, it is expected that some index files might cover Parquet files removed by compactions.

Although the indexing API is internally parallel, it should not be called on the same table column across multiple processes. While doing so will not violate any safety guarantees, the same Parquet files would be needlessly re-indexed multiple times.

One might argue that indexing every new Parquet file in the data lake is inefficient, as new Parquet files written by special processes such as Z-order or compaction could be more efficiently indexed by simply remapping the existing posting list values without recomputing the entire index. However, this procedure is significantly more complex with dangerous pitfalls: e.g. the original Parquet files that were compacted might have been removed, making the remapping impossible.

4.3.2 Searching a Rottnest Index

`search(table, snapshot, index_dir, query, K):`

table: the data lake table to search.

snapshot: the snapshot to search (Rottnest supports time travel).

index_dir: the location of the Rottnest index.

query: the query term. For full text indices this could be a keyword whereas for ANN indices this could be a vector embedding.

K: top-K results. This could have different meaning for exact match queries like regex (any K that satisfy predicate) and scoring queries like vector search (top K ranked based on score).

The search procedure follows these steps, with an example illustrated in Figure 4.4:

1. **Plan**: Rottnest first queries the data lake for the manifest list for the specified snapshot, which contains a list of Parquet files that make up the snapshot along with potential deletion vector files. Rottnest queries the metadata table in `index_dir` to determine which index files cover the Parquet files and which Parquet files have no index and must be brute-force scanned. In Figure 4.4, 09xf and ac02.index must be queried.

2. **Query Index**: Each Rottnest index file is queried independently in parallel for physical locations in the underlying Parquet files. The index files are queried on object storage directly, with optimization techniques described in Section 4.4.2. For example, 09xf.index might return [(a.parquet, 10), (a.parquet, 20)], where 10 and 20 denote locations in the Parquet file. Rottnest

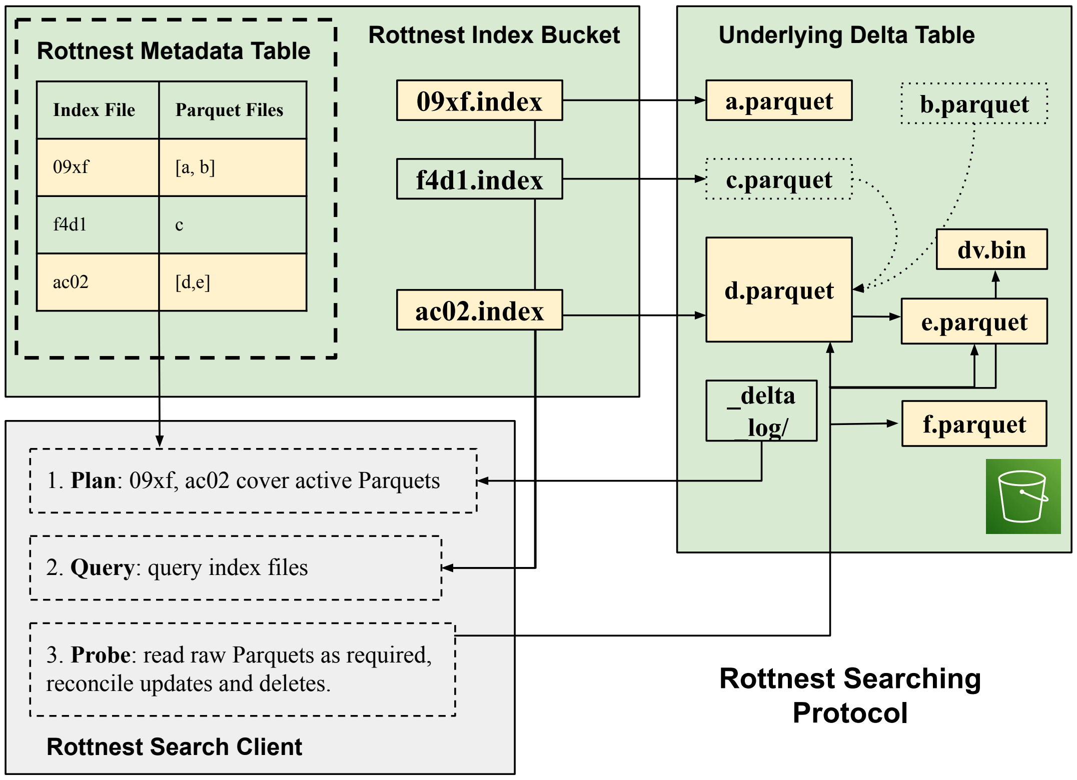


Figure 4.4: Rottnest Search Protocol based on the running example in Figure 4.3. Assume that after the `index` call, f.parquet is added to the table and is un-indexed.

indices are allowed to return false positives (e.g. bloom filter), so the top-K filter is not applied at this stage. Instead, we just filter out locations that correspond to Parquet files not in the specified `snapshot`. This step might fail. In this case, the entire search aborts and restarts.

3. **In-situ Probing**: The physical locations in the Parquet files are downloaded and scanned with the actual predicate to filter out false positives. Rottnest efficiently random accesses Parquet files, explained in more detail in Section 4.4.1. Deletion files, i.e. dv.bin, are applied if they exist. The unindexed Parquet files are only scanned if the filtered results are not sufficient to satisfy an exact-match top K query or for a scoring query, which must rank all data items.

The correctness of this procedure follows from the two invariants (existence and consistency) described at the start of Section 4.3. No row in the data lake will be "missed" since it is either covered by an index file or is in a Parquet file that would be exhaustively scanned. Different from the `index` API, the `search` API is read-only for the data lake and the Rottnest index. It is meant to be called in parallel by independent processes and concurrently with the `index`, `compaction` and `vacuum` APIs described in the next section.

Note that the search operation is *safe* with respect to compaction and vacuum operations of the underlying data lake. For example, assume a data lake compaction occured after the latest Rottnest indexing operation, e.g. replacing Parquet files a.parquet and b.parquet with c.parquet. In the Plan stage, Rottnest would recognize that c.parquet needs to be exhaustively scanned, because it is a "new Parquet" in the data lake, even though it was just a replacement for a.parquet and b.parquet. In the Query Index stage, index search results corresponding to a.parquet and b.parquet would be discarded. In the final stage, c.parquet would be exhaustively scanned. This suggests that data lake compactions result in less efficient, but correct, search if the Rottnest index is not updated.

In the case of a vacuum operation of the underlying data lake, old Parquet files are deleted. This is effectively a no-op with respect to Rottnest search efficiency because those files would be marked as skip any way in the first Plan stage.

It is important to note that even though the search operation is safe, it might not be *available*. If a concurrent vacuum operation on the Rottnest index starts and finishes between the Plan and Query Index stages of a search operation, a Rottnest index file might be deleted. In this case, the Rottnest searcher can simply abort and restart the entire search process.

4.3.3 Index Maintenance

To avoid querying many small index files as the data grows, Rottnest supports compacting indices similar to log-structured merged tree (LSM) mechanisms employed in databases such as RocksDB and Clickhouse [45,53]. Multiple small index files are compacted into larger files and older index files can be garbage collected. In Rottnest, the indices are compacted independently of the data maintenance process of the underlying data lake, which might perform its own LSM-style compactions.

Rottnest supports the following compaction API, which can be called by the indexer or in a separate process.
`compact(table, index_dir, index_args):`

table: the underlying data lake table.

index_dir: the location of the Rottnest index.

index_args: index-specific arguments that control how certain types of indices are merged.

The compaction process proceeds in three steps:

1. **Plan:** determine which index files to merge. In general, index files that cover small numbers of Parquet files or rows can be merged into larger ones, while it maybe less important (and more expensive) to merge indices that already cover a large number of files. The default behavior of Rottnest is to merge index files that is smaller than a configurable threshold in a bin-packing strategy.

2. **Merge:** build the merged index files. This step could be computationally intensive and might require reading the raw Parquet files. After the new index files are built, upload them to `index_dir`.

3. **Commit:** insert metadata about the merged index files into the metadata table.

It is important to note that the compaction process does not consult the log of the data lake at all, and is completely decoupled from the compaction process of the data lake itself. How exactly multiple index files are combined into one depends on the index type, some examples of which will be given in Section 4.5.2.

Similar to how compaction works in data lakes, Rottnest index compaction does *not* delete index files, which is the responsibility of a separate garbage collection process, commonly referred to as vacuum. There are multiple reasons an index file may be eligible for garbage collection:

- It was written by a failed indexer or compactor before a successful commit to the metadata table.
- It has undergone compaction, i.e. a new index file covers the Parquet files that this file covers.
- It points to Parquet files that are no longer part of a supported snapshot of the underlying data lake.

Rottnest offers the following API to remove those files:
`vacuum(table, index_dir, snapshot_id):`

table: the underlying data lake table.

index_dir: the location of the Rottnest index.

snapshot_id: the minimum snapshot of the underlying data lake to support time travel from, to determine the set of active Parquet files.

The vacuum process happens with three steps:

1. **Plan:** determine which index files in the metadata table to keep based on `snapshot_id`. Rottnest currently uses a simple heuristic: it first computes all Parquet files included in all the snapshots past `snapshot_id`. Then it greedily selects index files that cover the most number of active Parquet files to keep. The procedure stops when the number of covered Parquet files cannot be increased. This procedure maximizes the number of covered Parquet files while heuristically minimizing the number of index files.
2. **Commit:** Delete the rows in the metadata table corresponding to index files that are no longer necessary as determined by the last step.
3. **Remove:** Physically remove the index files from object storage that are no longer in the metadata table *and older than the index timeout*. Removing these "invisible" files require an expensive LIST operation against `index_dir`, which is acceptable because `vacuum` calls are not expected to be frequent or real-time.

We remove after commit in `vacuum` instead of commit after upload in `index` and `compact`. This ensures index files included in the metadata table are physically present to preserve the first invariant.

It is critical that vacuum uses the timestamps associated with objects to only remove objects older than the `index` timeout, since from its perspective there is no difference between index files written but not yet committed and index files that were written but failed to commit. Since the `index` operation has a timeout, `vacuum` knows files older than this timeout are in the second category and can be removed. Note that this timeout is against the object store's clock, which is valid because modern object stores provide strong consistency, and thus have a single global clock [35].

4.3.4 Invariants Proof of Correctness

Due to the loose synchronization between the index files, the underlying data lake, and the processes that modify them, Rottnest's protocols are carefully designed to ensure that data is either indexed correctly, or not at all, by maintaining the existence and consistency invariants. If M is the set of files referenced by the metadata table, and B is the set of files in the bucket, the following holds: **Existence**: all indexed files referenced in the metadata table are present in the object storage bucket (i.e. $\forall f \in M : f \in B$) We prove the result by induction. Initially, $M = \varnothing$ and the invariant trivially holds.

For the inductive step, assume the invariant holds before launching some number of index-modifying processes (either `index`, `compact`, or `vacuum`). There are three states these processes can be in:
`before_upload`, `before_commit`, and `during_delete`. First, notice that the indexing and compaction processes follow the same pattern of plan, upload, and commit. For both indexing and compaction in the `before_upload` state, both M and B are unmodified, so the invariant holds. In the `before_commit` state, B can only have grown with the new file f_{new}, and so $\forall f \in M : f \in B \cup \{f_{new}\}$ is true. Lastly, the commit will update M to contain f_{new}, and so $\forall f \in M \cup \{f_{new}\} : f \in B \cup \{f_{new}\}$. Note that concurrent updates do not change the nature of the proof, since updates to M are transactional and files uploaded to B are owned exclusively by the process.

For `vacuum`, suppose we decide to delete some files F. First note that a concurrent indexing or compaction operation may have introduced some files to B but not M. Since these operations are guaranteed to take less time than the global timeout (they abort otherwise), we know that it is not safe to delete files younger than the timeout, and any such files are skipped by the vacuum process. These files therefore cannot be in F, and so will never be deleted before they are written to M. By the induction hypothesis, in the `before_commit` state, the invariant holds. After transitioning to the `during_delete` state, M is updated to $M \setminus F$; since M shrank, $\forall f \in M \setminus F : f \in B$ holds. Following the `during_delete` state, B is updated to $B \setminus F$ and we have $\forall f \in M \setminus F : f \in B \setminus F$.
Consistency: an index file correctly indexes the associated Parquet files if they still exist. In other words, let d_f be the associated data file for index file f. Then $\forall f \in B : \neg exists(d_f) \vee indexes(f, d_f)$. Take an arbitrary Rottnest file $f \in B$. Once f is built, it correctly indexes d_f. Since both Rottnest and underlying data files are both immutable, the $indexes(f, d_f)$ will always be true unless either f or d_f is deleted. If f is deleted, then $f \neg \in B$, so the invariant holds. If d_f is deleted, then $\neg exists(d_f)$ is true, and so the invariant also holds.

4.4 Index Implementation

So far we have limited our discussion to maintaining consistency between inverted indices and physical locations. This section discusses our implementation decisions to achieve low indexing overhead and efficient querying. Rottnest queries have two main sources of latency. Section 5.1 discusses how we read the underlying data. Section 5.2 discusses how we optimize the layout of our indices to minimize round trips to object storage.

4.4.1 In-situ Querying

It might seem surprising from our discussion in Section 4.1 that keeping the original data in Parquet on object storage could lead to efficient querying. A typical layout of a Parquet file is shown in Figure 4.5. Since Parquet writers typically write 128MB row groups, and we are interested in

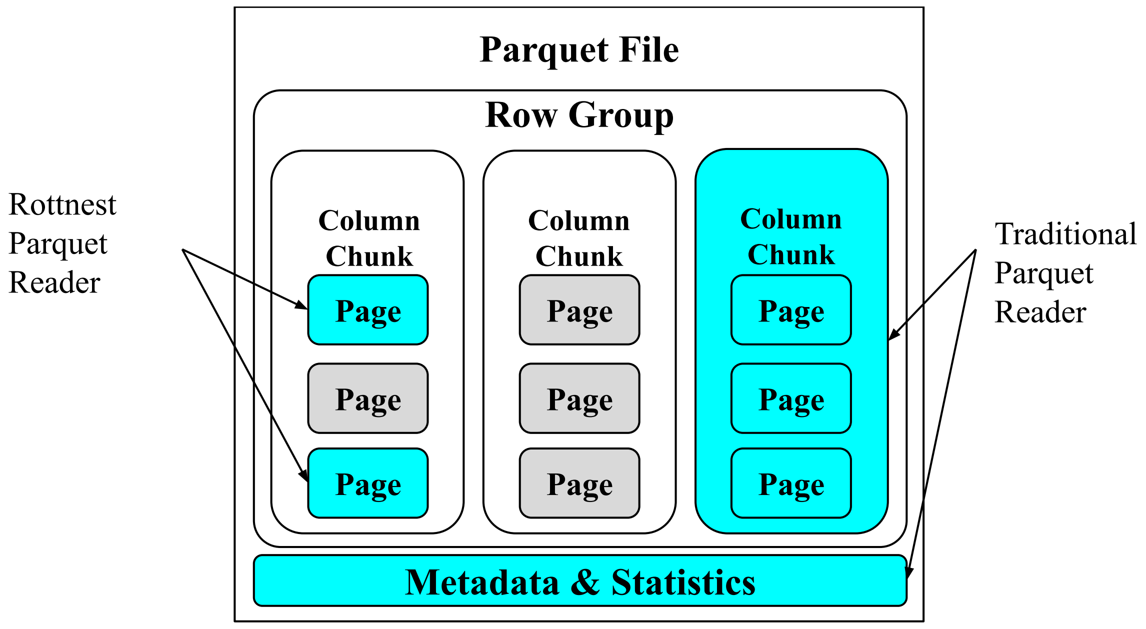


Figure 4.5: Traditional Parquet readers read entire column chunks. Rottnest's reader reads individual pages, and notably bypasses the file metadata.

indexing wide columns (vectors and text), the row group's space is dominated by our column's column chunk (typically 100 of 128 MB goes to our column). [5] For highly selective search queries, reading and decompressing 100MB of data from object storage to retrieve just a few rows is not efficient.

To mitigate this issue, we observe that the minimal access granularity in a Parquet file is actually a *data page*, whose size is independent of the row group size. Typically, the physical size of a data page is equal to the compressed size of 1MB of raw data, which is around a few hundreds KBs for text or vector data types. We will show in Section 4.5.5 that reading at this granularity is as efficient as using custom data formats like Lance [87].

Similar to NoDB which maintains *position zone maps* on raw data [14], Rottnest maintains *page tables* that associate a unique ID for each data page to the offsets and sizes of the data page. Rottnest's indices are built at the granularity of these pages. In other words, the posting lists do not point to individual rows but to data pages. While this adds additional filtering work at query time and might complicate the design of indices which rely on posting list intersections (e.g. BM25), it significantly reduces the index storage footprint and speeds up index construction.

The dominant part of the index is typically the posting lists, which appear in both full text search indices and vector indices (via some variant of K-means). For instance, despite using state-of-the-art

[5] This is an inherent flaw in Parquet's design, because all column chunks in a row group must have the same number of rows.

compression algorithms like RoaringBitmaps [91], ElasticSearch's posting lists often eclipse the size of the underlying compressed text. However, since Rottnest's posting lists store just data page IDs instead of document IDs, the dynamic range of the posting list elements is an order of magnitude smaller, greatly improving the performance of compression algorithms, making Rottnest's posting lists of similar size to the compressed underlying data.

4.4.2 Optimizing for Object Storage Accesses

Object storage is a high-latency but high-throughput storage medium that favors large sequential range requests issued in parallel (access *width*) over sequences of small requests (access *depth*).[6]

The straightforward approach is to take the in-memory data structure, serialize and compress it, and then upload it to object storage. To query the data structure, simply download and decompress it in memory. We find compression to be a natural step after serialization, because it usually drastically reduces both the index file's size and the bytes that must be read upon query. The IO benefit almost always dominates the added compute cost of decompression. While this approach leads to large sequential reads and adequate parallelism assuming simultaneous querying of multiple index files, it can be wasteful for random-access indices where most of the data structure is not accessed.

An opposite approach could consist of "memory-mapping" the data structure to object storage, where memory accesses are directly translated to object store requests. This approach has the benefit of reading only the bytes required, however it could lead to long chains of dependent object-storage requests. In addition, it foregoes the compression benefits offered by the former approach.

An intermediate approach that we employ, which we term *componentization*, consists of breaking up a data structure into several serializable components. Each component is then compressed and concatenated to form the index file, which also contains an offset array of the location of each component. Each data structure access only reads the components it needs, while the total number of dependent requests is reduced because each component could capture multiple requests.

An example of this approach applied to a binary search tree (BST) is illustrated in Figure 4.6. Each query must read the "root" component and one leaf component. The other three leaf components are not read. The memory-mapped approach for each query would have required four sequential requests, while this approach requires only two.

The key observation that enables componentization is that most data structures employed in indices have some degree of "access locality". This approach would not work if after an access into the data structure, the next access address is completely random or data dependent. In these cases, alternative data structures might have to be considered.

[6]For a more in depth discussion of object store performance characteristics, we refer the reader to the original motivations of Delta Lake [27] and the AWS Performance Guide [21].

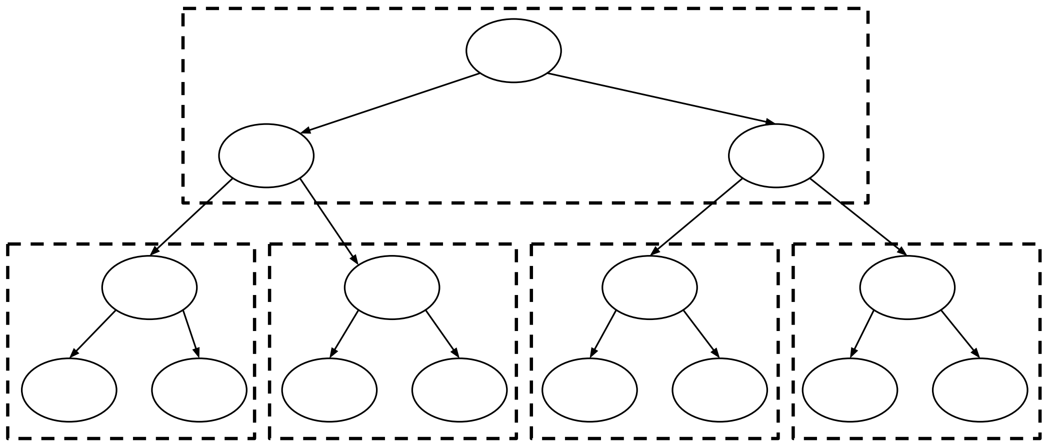


Figure 4.6: Breaking a BST into serializable components.

4.5 Evaluation

In Figure 4.2, we presented intuition for how Rottnest compares to two other approaches: *copy data* into a dedicated system or *brute force* scan the data lake. We now follow up with a much more quantitative evaluation framework that seeks to answer under what exact conditions is one approach better than another to ground our evaluation results. We apply this framework to three Rottnest indices we constructed for three example applications: UUID, substring and vector search.

4.5.1 Evaluating Rottnest

We assume we are operating in a regime where we are not latency constrained in Figure 4.2. While the *minimum latency threshold* of an approach is an important metric, a better way to evaluate these three approaches beyond the threshold is using cost: the *total cost of ownership (TCO)* of the system under a fixed query load and operating duration. In this situation, comparing query latency is less appropriate since the brute force approach can trivially reduce query latency by scaling search horizontally.[7]

Since all three approaches can be more efficient if the query only has to search part of the data lake, e.g. due to a filter on a structured attribute like timestamp, we consider the cost per *normalized query*, where the brute force approach must scan the entire dataset. Note that Rottnest can leverage structured filters by building indices on different partitions of the data clustered by the structured attribute. Indexing systems like ElasticSearch or vector databases also have optimizations for this use case [110, 140].

[7]In practice, the scaling is not perfect and the cost can still increase. We will examine this later in Section 4.5.3.

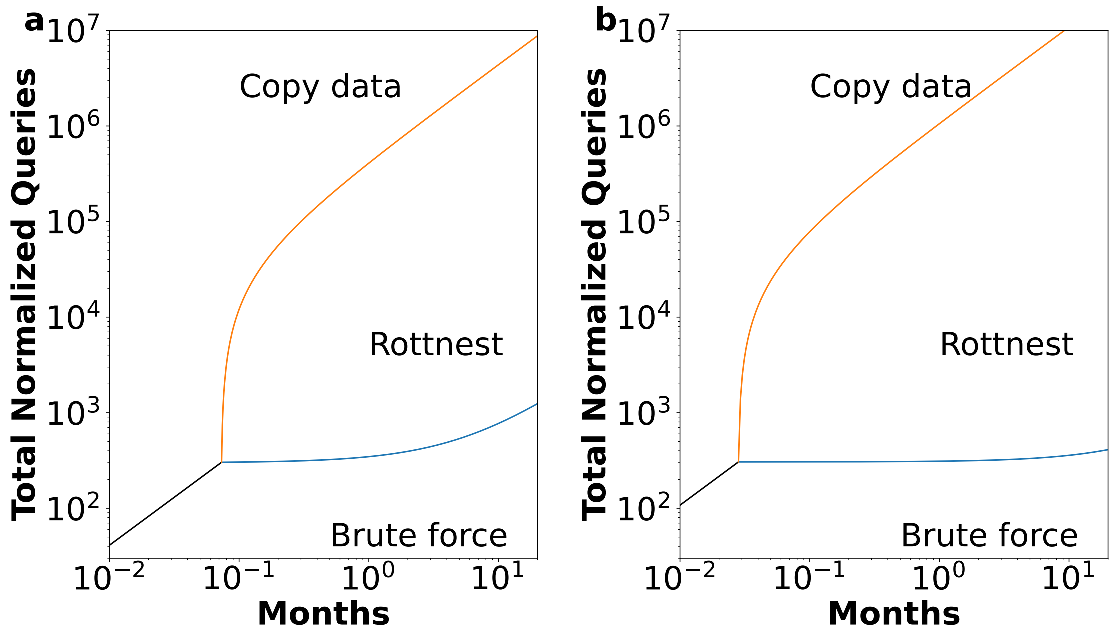


Figure 4.7: Phase change diagrams for a) Substring search and b) UUID search. Note log-log axes. Explained in Section 4.5.4.

To compare the TCO of these three approaches, we can plot a quadrant with the total number of normalized queries on the y-axis and the number of months we are operating the system on the x-axis. For a particular point (months, queries) on this quadrant, we can estimate the TCO of each of the three approaches as $index\_cost + cost\_per\_month \times months + cost\_per\_query \times queries$:

- **Copying data** into an dedicated system typically incurs a high cost per month for a cluster of always-on servers. On the other hand, the indexing and query cost can be folded into this constant monthly operating cost. $TCO = cpm\_i \times months$, where cpm_i is the cost per month.
- **Brute force** incurs no indexing cost and a very low cost per month (S3 storage of compressed data). However it has very high $cost\_per\_query$: $TCO = cpm\_bf \times months + cpq\_bf \times queries$, where cpm_bf and cpq_bf represent the cost per month and per query respectively.
- **Rottnest indices** incur a one-time index cost, relatively higher cost per month (to store and maintain the index structures) but a much lower $cost\_per\_query$ compared to brute force. $TCO = ic\_r + cpm\_r \times months + cpq\_r \times queries$, where ic_r denotes the index cost.

Using this model we can plot a "phase change" diagram that indicates the most economical solution for each point in the quadrant, a couple examples of which are shown in Figure 4.7. The lines indicate boundaries between regions where a different approach is optimal in terms of TCO.

This phase change graph allows a practitioner to easily figure out the most economical approach based on the estimated usage characteristics of the search workload. For example, at 10 months and 10^4 total normalized queries, Rottnest is the most cost efficient approach for substring search.

Rottnest typically becomes most economical when:

1. The dataset needs to be queried more than some minimum number of times to amortize the index cost.

2. The number of queries is large enough that brute force is too expensive but not large enough to justify copying the data into an always-on index.

The parameters *cpm_i*, *cpm_bf*, *cpq_bf*, *ic_r*, *cpm_r* and *cpq_r*, which directly determine the shape of the graph, are generally dependent on the search workload and data distribution. We will discuss how changing them affects the phase change diagram in Section 4.5.6.

4.5.2 Example Rottnest Indices

In this section, we describe example Rottnest indices for the search workloads described in Section 4.1: UUID, substring and vector search. As discussed, current data lake query engines struggle to perform well on these workloads. While the Rottnest indices are based on classic indices such as binary tries and FM-indices, we explain how to adapt them to object storage and to support the efficient merging required for compaction.

High-cardinality UUIDs

We design an index to enable fast exact UUID matching using a binary trie, where each UUID defines a unique path through the trie structure. Following the design principles outlined in Section 4.4.2, we decompose the binary trie into modular components as illustrated in Figure 4.6.

To optimize storage efficiency, the binary trie indexes only the minimum number of bits required to uniquely distinguish each UUID. Rather than computing the exact longest common prefix (LCP) for each UUID, we estimate the required indexing depth based on the entropy derived from the alphabet size and corpus size. Since indices may be merged dynamically, we cannot determine precise LCP values in advance. To accommodate this uncertainty and potential estimation errors, we index up to 8 additional bits beyond our entropy-based estimate for each UUID, while allowing leaf nodes to reference multiple UUIDs when this extra depth proves insufficient for unique identification.

The Rottnest index framework naturally accomodates false positives that may occur due to insufficient indexing depth, as even true positive matches require subsequent full page scans of the corresponding Parquet pages. When false positives occur, additional Parquet pages must be read and scanned, resulting in increased I/O overhead and slower query performance downstream.

We implement a further optimization by replacing the trie's first 8 levels with a direct lookup table, thereby reducing the number of sequential memory accesses required during traversal. The fundamental principle guiding this approach is that optimal binary trie performance requires trie depth to closely match the expected longest common prefix among all indexed keys, minimizing false matches during lookup operations.

Exact Substring Matching

We employ the FM-index based on the Burrows-Wheeler Transform [41, 60] with a sampled suffix array. The data structures are adapted to object storage with the componentization approach described in Section 4.4.2. To merge indices, we employ the technique described in [72] with bounded interleave iterations. We will explain the FM-index in much more detail in the following chapter.

Vector ANN Index

Vector indices are typically graph based (e.g. Vamana, HNSW) or clustering based (IVF-PQ) [77, 79, 97]. Our ANN index is based on IVF-PQ. While graph-based indices provide superior recall-speed tradeoffs in memory or even on disk, we find the graph structure incurs a long sequence of dependent requests and is not easy to break down into non-overlapping components. As a result we opt for a simpler index based on IVF-PQ and adjust recall by increasing the *nprobe* and *refine* parameters.

Evaluation Methodology

We use AWS OpenSearch for the copy data approach for substring and UUID search and LanceDB for vector search. [87]. We use PySpark on AWS EMR for the brute force approach.[8] We use the C4 dataset included in FineWeb for substring search, which is 304GB compressed in Parquet format, containing around 0.8 trillion characters [111]. We use 2 billion randomly generated 128-byte hashes for UUID search, 131 GB in Parquet format. We use the SIFT dataset for vector search containing 1 billion 128 dimensional vectors, which is 137 GB compressed in Parquet format [94].

For *cpm_i*, we include the EBS cost required to replicate the primary index three times for OpenSearch or LanceDB as well as three r6g.large instances. *cpm_r* and *cpm_bf* are computed based on the cost of storing the raw data and the raw data plus the Rottnest index on S3 respectively. *cpq_bf* and *cpq_r* are computed from query latency times the hourly cost of the EC2 instances on which the queries are executed. The indexing cost *ic_r*, includes both the EC2 instance cost for Rottnest to compute initial indices and adequate compaction.[9] Technically, *ic_r*, *cpq_bf* and *cpq_r*

[8]Vector search is implemented with a UDF using the mapInArrow API.

[9]Compaction could also be counted in *cpm_r*. For the append-heavy datasets here, data no longer needs to be compacted once they are in adequately sized indices, making it more appropriate to include compaction cost in the upfront indexing cost.

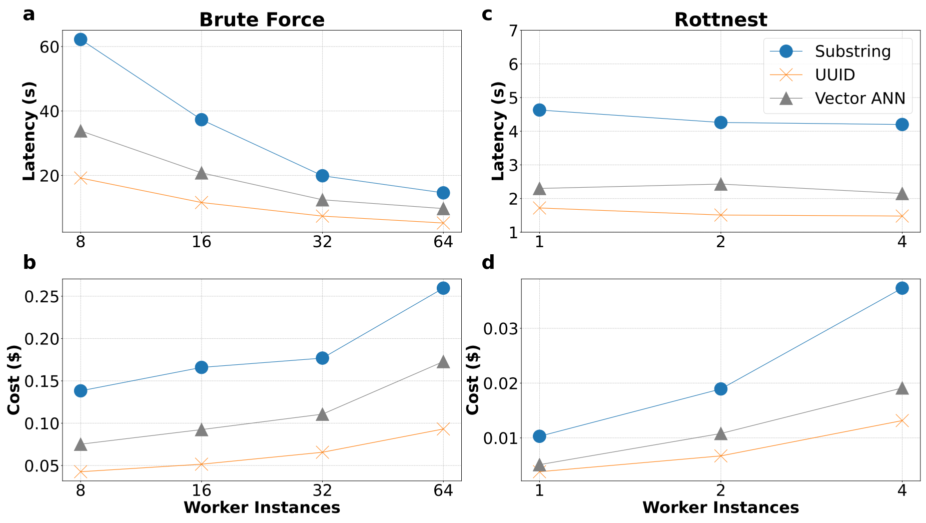


Figure 4.8: Brute force approach latency (a) and cost (b) scaling with cluster size. Rottnest latency (c) and cost (d) scaling with cluster size. Worker instance used is r4i with 16 vCPUs. The graphs show that both Rottnest and brute force approaches cannot achieve arbitrarily low latency by simply scaling up compute – there is some lower limit due to the need for object storage access.

should include the cost of S3 requests. In practice, we find them eclipsed by compute resource costs so focus on the latter for the evaluation.

4.5.3 Minimum Acheivable Latency and Maximum Tolerable Latency

Before we address TCO, we seek to determine the minimum achievable latency of Rottnest and the brute force approach for the three applications. We examine how horizontally scaling the number of machines impact the latency and the query cost, as described in the last section, in Figure 4.8.

From Figure 4.8a and 4.8b, we see that Spark is fairly horizontally scalable up to 32 worker instances across the three query types. Scaling to 64 workers leads to a marked decrease in latency improvement and therefore a large increase in cost per query. The latency at 64 workers can be taken as an estimate of the minimum achievable latency where brute force becomes a viable approach.

Rottnest is not easily horizontally scalable, since it is often latency bound instead of throughput bound: as discussed in Section 4.4.2, we find ourselves bottlenecked by the *depth* of our object storage reads instead of the *width*. As a result, Figure 4.8c and 4.8d show the latency barely improving with more searchers, while the cost almost linearly increases. Indeed, Rottnest is designed to be operated in practice with a shared-nothing architecture.

We note that for all three query types, Rottnest's latency on one worker still outperforms brute force latency on 64 workers by a large margin: 4.3x for substring and UUID search, and 5.4x for vector search. This means Rottnest has a much lower minimum achievable latency: 4.6s for substring

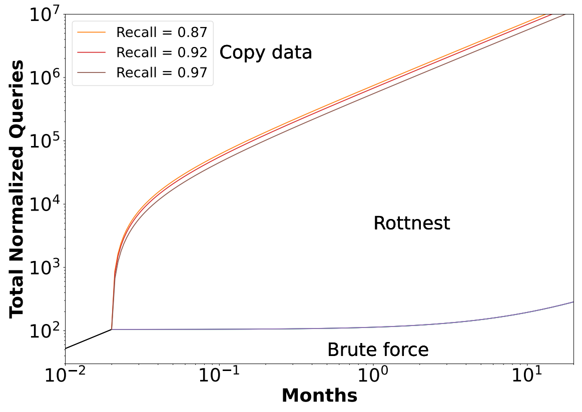


Figure 4.9: Phase change diagrams for vector search at different recall targets. Note log-log axes.

search, 1.7s for UUID search and 2.3s for vector search.

These results mean that if the system requires sub-second latencies, Rottnest and brute force and unfortunately unsuitable.

4.5.4 Total Cost of Ownership

We now apply our TCO evaluation framework described in Section 4.5.1. We use 8 worker instances for brute force and a single instance for Rottnest. Note they are the most cost efficient configurations explored in the last section for both approaches. We separate the three applications into *exact queries* (substring and UUID) and *scoring queries* (vector), where the evaluation is complicated slightly by recall tradeoffs.

Exact Queries

We first evaluate the exact match queries, i.e. UUID and substring search. The phase diagrams are plotted on Figure 4.7. We see that the point at which Rottnest becomes a competitive option starts very early for both applications (2 days for substring search and less than 1 day for UUID search). As the number of months increase, the range of total query numbers in which Rottnest is most economical grow to span more than 4 orders of magnitude: from around 8×10^2 to 4×10^6 total queries at 10 months for substring search and from 3×10^2 to 10^7 for UUID search.

We see a curvature up in the boundary between Rottnest and brute force for substring search since the Rottnest indices are almost as large as the compressed Parquets in this case. This makes brute force increasingly economical as the operating duration increases. For the UUID search, the indices are much smaller, so the boundary stays flat. This behavior will be discussed in detail in Section 4.5.6.

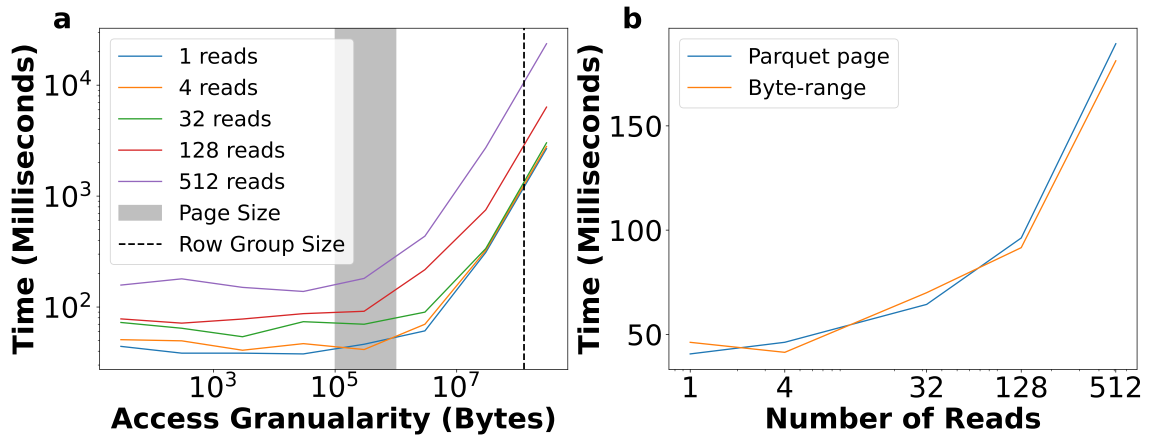


Figure 4.10: Parquet reading benchmarks showing how a) read latency increases with read granularity at different number of concurrent reads and b) the latency of reading 300KB byte ranges compares to reading real Parquet pages.

Scoring Queries

The evaluation approach for vector search needs to be modified since query cost can be traded off with recall for Rottnest. We assume that changing the recall target has negligible effect on the TCO of the other two approaches. This is definitely the case for brute force, where the recall is always perfect. This is less true for copy data approach, where the recall target could degrade the throughput of the vector database which might require more servers to hit a QPS target. We ignore this consideration here, which makes the copy data baseline stronger in the evaluation presented.

The Rottnest vector index is based on IVF-PQ [9,54]. We tune the *nprobe* and *refine* parameters to hit different recall@10 targets: 0.87, 0.92 and 0.97. The former controls how many centroids to probe and PQ vectors to rank, whereas the latter controls how many full precision vectors to download and rerank. Increasing these parameters improves recall but also increases search latency and *cpq_r*.

We show the phase diagrams corresponding to the different recall targets in Figure 4.9. In our experiments, a recall target of 0.97 leads to a higher search latency, and thus *cpm_r*, 35% worse compared to a recall of 0.87. However this difference barely changes the area Rottnest wins in the log-log plot, which covers around 4 orders of magnitude at 10 months. Indeed, given the large orders of magnitude differences between *cpm_i* and *cpm_r*, the small changes in *cpq_r* does not move the boundary significantly. Concretely, this means building a Rottnest index is most likely still a good decision if recall target changes due to business requirements or if different queries have different recall requirements.

4.5.5 In-situ Querying

A key decision point for Rottnest is to read raw data from the underlying Parquet files, instead of copying the raw data into a custom storage format. This decreases both *ic_r* and *cpm_r* as it reduces storage requirements. The Rottnest index is typically much smaller than the compressed data itself, so storing a copy of the data would multiply the storage overhead several fold. In Figure 4.11, we show the effect including a copy of the data would have on the phase diagram of the UUID search. On longer time horizons at around 10 months, it reduces the range of total queries where Rottnest is more cost-effective than brute force by a few times and the cost benefit compared to brute force in general.

The catch is that keeping the data in Parquet makes querying more challenging, as open source Parquet reader implementations cannot efficiently perform random access on this data, increasing *cpq_r*. In Figure 4.11 we show that without a custom reader, Rottnest becomes less efficient than the copy data approach over several orders of magnitude. Rottnest resolves this by designing its own optimized Parquet reader that reads only the required column pages (∼300KB) vs entire row groups (∼128MB).

Compared to an ideal custom format that allows the reader to fetch only the bytes necessary for a data item (typically 0.1-4KB) without decompression, our Parquet reader has a read granularity of 300KB and must decompress the whole read to fetch any item. We experimentally validate that our Parquet reader makes our in-situ querying likely as efficient as using this ideal custom format. We first show in Figure 4.10a that byte-range read request latency to S3 is stable in terms of read granularity until around 1MB, at which point it increases linearly with the read size. This observation holds for different numbers of concurrent reads from 1 to 512. While the size of Parquet row groups puts it in the throughput bound regime, the size of Parquet pages puts it squarely in the latency bound regime. Concretely, this means using a custom storage format to perform more granular reads is unlikely to result in improved performance. In addition, we show in Figure 4.10b that there is little difference in terms of performance between reading 300KB byte ranges and reading and decoding actual Parquet pages in our Parquet reader, showing that decompression overhead is not a concern.

To further evaluate this key design choice, we directly compare Rottnest's query performance to LanceDB cold cache mode which uses its own custom Lance format. In contrast to the LanceDB configuration used to benchmark the copy-data approach above where the index is kept in memory, we keep the index on S3 and query it directly similar to Rottnest, using optimized configurations tuned by a core LanceDB maintainer. Rottnest achieves comparable search latency at all recall targets: 2.09s (Rottnest) vs 1.90s (Lance) at 0.87, 2.30s vs 1.94s at 0.92 and 2.81s vs 2.72s at 0.97. This experimentally validates that using a custom format is unlikely to significantly improve query performance compared to in-situ querying with Rottnest's custom Parquet reader.

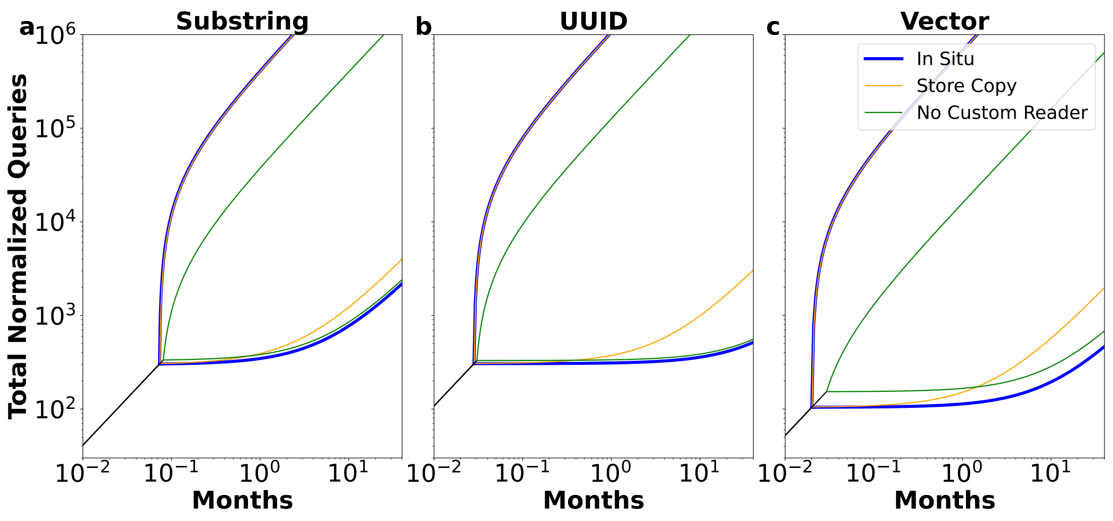


Figure 4.11: Changes to the phase diagram if Rottnest keeps a copy of the data in custom format or if it did not use an optimized custom Parquet reader.

4.5.6 Sensitivity Analysis

Parameter Robustness

The last section has demonstrated that changing *cpq_r* and *cpm_r* have dramatic effects on the phase diagram. In this section, we systematically examine the impact of changing *cpq_r*, *ic_r* and *cpm_r* on the phase diagram to understand the effects of optimizing each. Figure 4.12 demonstrates how the phase diagrams shift for vector search (0.92 recall) when each of these parameters are multiplied by the shown factor. For *cpm_r*, we show the result of scaling $cpm\_r - cpm\_bf$, or just the storage cost associated with the Rottnest index files. Two observations:

1. Decreasing Rottnest search latency (*cpq_r*) makes it more competitive against copy data, with virtually no benefit against brute force. Decreasing the Rottnest index size (*cpm_r*) does exactly the opposite.

2. Reducing the indexing cost (*ic_r*) reduces the minimum operating months at which Rottnest becomes worthwhile compared to the other approaches. On the other hand, it does little to the asymptotic boundary between Rottnest and the other two approaches at longer operating time horizons.

These observations inform how optimizations in Rottnest directly benefit different classes of use cases: making the search faster benefits high query load applications; making the index smaller benefits low load applications and making the index construction cheaper benefits applications with

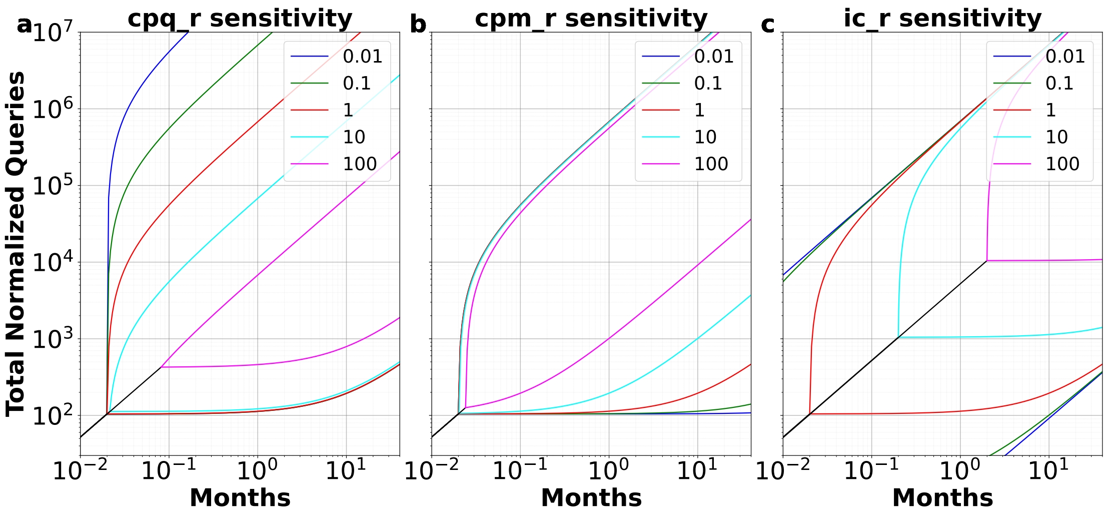


Figure 4.12: Sensitivity analysis of *cpq_r*, *ic_r* and *cpm_r* for vector search application at recall 0.92. Contours indicate phase diagrams if each of the parameter is multiplied by the denoted factor. The actual diagram is in red.

short operational lifetimes.

Dataset Size Robustness

So far, all the parameters are computed based on fixed dataset sizes, as described in Section 4.5.2. While the key parameters *cpm_i*, *cpm_bf*, *cpq_bf*, *ic_r*, *cpm_r* and *cpq_r* are evidently dependent on the *data distribution* in complex nonlinear ways[10], most are almost perfectly linearly correlated with *dataset size* assuming the same data distribution, which would imply no change in the phase diagram.

While *cpq_r* generally scales with the number of index files queried, which generally increases linearly with dataset size, Rottnest compactions could greatly reduce this number to dramatic effect as seen in Figure 4.13. For the case of UUID search, we also see nonlinear scaling due to AWS list request throttling issues. Post compaction, the Rottnest search latency is effectively constant irrespective of the dataset size, which means that as data volume increases, *cpq_r* stays relatively constant while all other parameters increase linearly, making Rottnest more attractive against the copy data approach as shown in Figure 4.12a.

[10]For example, entropy influences compression efficacy for text datasets.

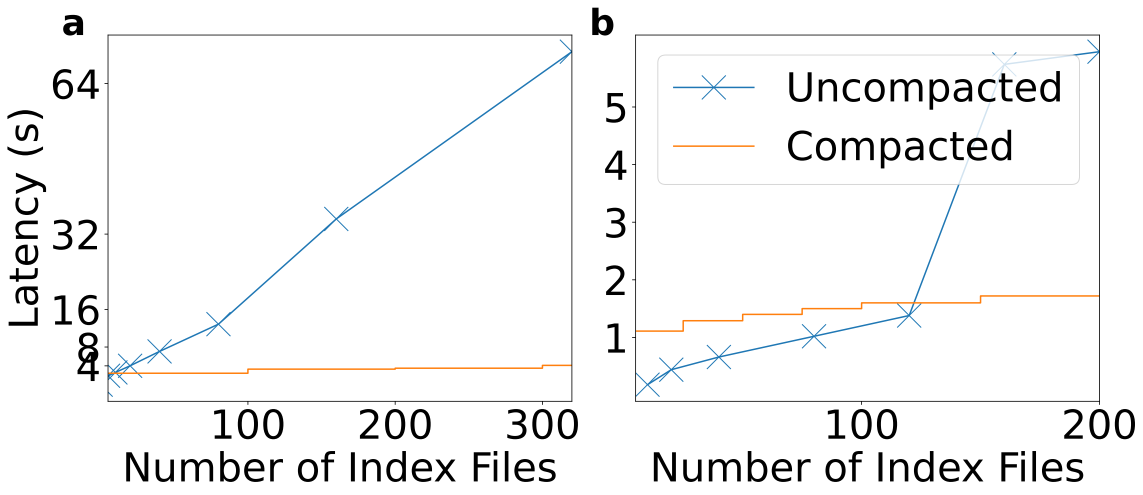


Figure 4.13: Search latency on uncompacted vs. compacted index files for a) substring (100x compaction factor) and b) UUID search (25x compaction factor). Compaction greatly reduces search latency when there is a large number of index files.

Throughput Limitations

While our evaluation framework covers concerns such as search latency, total operating cost and operating duration, we did not discuss the maximum throughput supported by the three approaches. While the copy data approach is typically bottlenecked by the disk IOPs and CPUs of the dedicated servers, Rottnest and the brute force approach are bottlenecked by S3's limit of 5500 GET RPS per prefix. While the number of requests Rottnest makes is heavily dependent on the query[11], this typically caps Rottnest's QPS at 10-100. However, from Figure 4.7 and Figure 4.9, Rottnest already underperforms copy-data approach at these QPS levels (10QPS $= 2.52 \times 10^7$ total queries at 10 months). As a result, these throughput limits do not significantly change the conclusions drawn here.

4.6 Discussion

Current cloud OLAP systems fall into two main categories: compute-storage integrated and disaggregated systems. Integrated systems like Clickhouse, ElasticSearch, and Splunk operate as server clusters with data sharded across RAM/SSDs [46, 57, 130]. While these systems enable millisecond-latency queries through warm storage, they require long-running servers and data replication, resulting in high operating costs and scaling challenges [80].

[11]Exact queries like UUID and substring search send a couple hundred requests to the Rottnest index files but only up to a dozen requests to the underlying data lake, whereas vector queries can send hundreds of requests to both the index files and the data lake.

These limitations led to compute-storage disaggregated systems like Snowflake, BigQuery, and data lakehouses [17, 29, 48, 62, 99]. These systems query data stored in object storage with horizontally scaled workers. While offering serverless pricing and unlimited scalability, their limited indexing capabilities (such as Parquet's block min-max indexes [22] and Grafana Loki's attribute-level indexing [67]) lead to higher per-query costs due to extensive data scanning.

Rottnest represents an emerging third category: indexed disaggregated systems, which use object-store-native, domain-specific indices to accelerate queries. Similar approaches include Indexed DataFrames for Spark joins [135], Apache Hudi's B-Tree indices [149] and Hyperspace's external indices [112, 113]. Rottnest addresses certain limitations in previous systems, while focusing on search workloads. For instance, Microsoft's Hyperspace system is limited to cloud storage backends that support atomic compare and swap. Similarly, Apache Hudi's indexing subsystem is tightly integrated with the data lake's metadata table, while Rottnest can be bolted on to any data lake [131].

We show how Rottnest is the most cost-effective solution across several orders of magnitude of total query load across a wide range of operating durations for the exemplar applications listed in Section 4.1. While Rottnest incurs a couple seconds minimum latency, we believe most of these applications are tolerant to search latency as they are either human interactive, or the search latency is eclipsed by other sources of latency, such as LLM generation latency (on the order of 10 seconds for 500 tokens [114]).

Besides the TCO considerations presented, there are practical benefits to deploying Rottnest in the aforementioned, typically spiky workloads. Dedicated clusters like ElasticSearch take minutes to scale up and down, making it difficult to rightsize them for the load. Brute force approaches requiring distributed compute is hard to deploy in practice: either a shared cluster is used or each query spins up its own cluster. The former leads to poor performance isolation between different user queries and the latter can incur significant spinup overheads.

Since Rottnest is designed to provide acceptable search latencies from *one* search instance with object storage as the only shared state, it easily supports scalable shared-nothing deployment architectures with serverless functions like AWS Lambda or client machines (e.g. a data scientist's devbox inside the organization's cloud account), greatly reducing infrastructure complexity and cost. We have open sourced Rottnest[12] to facilitate further research.

[12] https://github.com/marsupialtail/rottnest

Chapter 5

Storage: Detailed Example for Observability Data

5.1 Motivation

In Chapter 4, we introduced Rottnest, a system for building external indices on data lakes to support diverse search workloads. We demonstrated its effectiveness across three representative applications: high-cardinality UUID filtering, substring search, and vector embeddings. Among these, substring search on machine-generated logs stands out as a particularly compelling use case that warrants deeper exploration.

Log analytics represents one of the most challenging search workloads in modern cloud infrastructure. Organizations generate terabytes of logs daily from their distributed systems, microservices, and cloud applications. These logs must be searchable for troubleshooting incidents, debugging performance issues, and conducting security audits [50, 56, 129]. The search patterns are inherently unpredictable—an engineer might search for a specific Kubernetes pod ID, a partial IP address, or a stack trace fragment. This variety makes traditional indexing approaches that rely on predefined sort orders ineffective.

The current landscape of log management systems exemplifies the exact trade-offs we discussed in Chapter 4. On one end, systems like ElasticSearch, Splunk, and DataDog provide millisecond-latency searches through sophisticated full-text indices, but require expensive always-on infrastructure and can consume storage approaching the size of the raw logs themselves [107, 118, 129, 147]. On the other end, brute-force approaches that store compressed logs in systems like Scalyr, Loki, or simply as Parquet files on object storage achieve an order of magnitude reduction in storage costs but make interactive searching prohibitively expensive [67, 122, 123].

This dichotomy makes log analytics an ideal candidate for Rottnest's indexed disaggregated approach. Logs exhibit several characteristics that align perfectly with Rottnest's design principles:

- **Append-heavy workload**: Logs are rarely updated after creation, making lazy indexing particularly suitable.
- **Moderate query frequency**: Most logs are queried only during incidents or investigations, falling squarely in Rottnest's optimal TCO range.
- **Tolerance for multi-second latency**: Interactive debugging can tolerate search latencies of several seconds, well above Rottnest's minimum threshold.

In this chapter, we present LogCloud, a specialized Rottnest index implementation optimized for log data. While Chapter 4 demonstrated the general applicability of external indices across diverse workloads, LogCloud shows how domain-specific optimizations can push the boundaries of what's possible with object-storage-native indices. Building on the general framework presented by Rottnest [112, 149], LogCloud addresses two key technical challenges unique to log search:

First, logs contain a mix of static templates and dynamic variables (like pod IDs or request identifiers), allowing for specialized compression techniques that dramatically reduce the data volume to be indexed. We leverage LogGrep's state-of-the-art log compression to break logs into template and variable components [147], then construct inverted indices specifically on the variables.

Second, the substring search requirements of log analytics demand more sophisticated index structures than the binary tries used for UUID search in Chapter 4. To address this challenge, we employ the FM-index based on the Burrows Wheeler Transform (BWT). While prior research has explored various FM-index implementations, they have focused primarily on disk/RAM settings [12, 41, 61, 65, 69], which are unsuitable for object storage's high-latency characteristics. We propose a novel implementation optimized for object storage that significantly improves search latency while maintaining compression efficiency.

Through LogCloud, we demonstrate that by combining Rottnest's external indexing approach with log-specific optimizations, we can achieve interactive search performance comparable to dedicated log services while maintaining less than 10% of their storage footprint. This results in up to 10x total cost of ownership savings for large-scale log datasets, validating Rottnest's promise of enabling new workloads on data lakes without sacrificing the benefits of disaggregated storage.

5.2 Background

5.2.1 Inverted Indices after Log Compression

Recent work like CLP and LogGrep has demonstrated that logs are highly repetitive and can be effectively compressed by exploiting static and runtime patterns, as shown in Figure 5.1 [118, 147, 148]. Almost all logs can be decomposed into repeated *templates* and *variable* components (e.g., request

```
2018-06-27 00:00:07,771 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode:
Sending heartbeat with 1 storage reports from service actor: Block pool
BP-596011733-172.18.0.2-1528179317196 (Datanode Uuid
c3bb40ae-c869-4ea0-ad0a-94f4f39bb5c6 ) service to master/172.18.0.2:8200
```

Figure 5.1: Logs are typically made up of fixed *templates* and changing *variables*, which are highlighted in yellow. Logs that do not fit into common templates are called *outliers*.

IDs or pod names, typically long pseudo-random alphanumeric URIs). LogCloud uses LogGrep to first decompose logs and indexes the variable components only. The templates, typically small in size, can be brute force searched.

To build the index itself, we leverage an inverted index structure that maps each variable to a *posting list* - an ordered collection of document IDs and positions where the token appears. These tokens are stored in a *term dictionary*, a collection of all unique tokens with pointers to their posting lists. The challenge lies in efficiently managing the *secondary index* needed to quickly look up tokens in this term dictionary, which can grow to multiple GBs when dealing with URI-style variables. Two popular approaches for this secondary index are finite state transducers (FST), used by ElasticSearch, OpenSearch, and M3DB [57,95,101,107], and sorted string tables (SSTables) [109], adopted by systems like Cassandra and Quickwit [52,85,116].

5.2.2 Challenge: Substring Searches on URIs

While both FSTs and SSTables enable efficient prefix (query$*$) and exact-match string queries on these variable tokens, they lack support for efficient substring ($*$query$*$) searches. Some systems (Quickwit, Cassandra) simply do not support substring search, while others (ElasticSearch) cannot efficiently use the secondary index and perform expensive scans of the term dictionary [57,85,116].

In search engine use cases, a term dictionary scan is acceptable, as the size of the language vocabulary does not grow linearly with the amount of text being indexed. However, as shown in Figure 5.1, the term dictionary here consists of unique resource identifiers (URIs)[1] whose number increases linearly with the size of logs being indexed. We will show in Section 5.2 that scanning this term dictionary can be very expensive for larger datasets.

Substring searches are critical for observability and cybersecurity use cases [33, 80, 128]. For example, an engineer troubleshooting a service outage might search for a partial URI '172.18.0.2' embedded in a larger URI, as shown in Figure 5.1, to correlate across log sources. As a second example, a security analyst investigating potential threats needs to search for partial IP addresses or domain fragments in network logs to identify suspicious traffic patterns (e.g., searching for "10.0.0." to find all matching IPs, or ".xyz" to detect traffic to suspicious top-level domains). In addition to these practical use cases, users often rely on substring queries rather than prefix or exact matches

[1]e.g. BP-59601* in Figure 5.1, **not** URLs on the web!

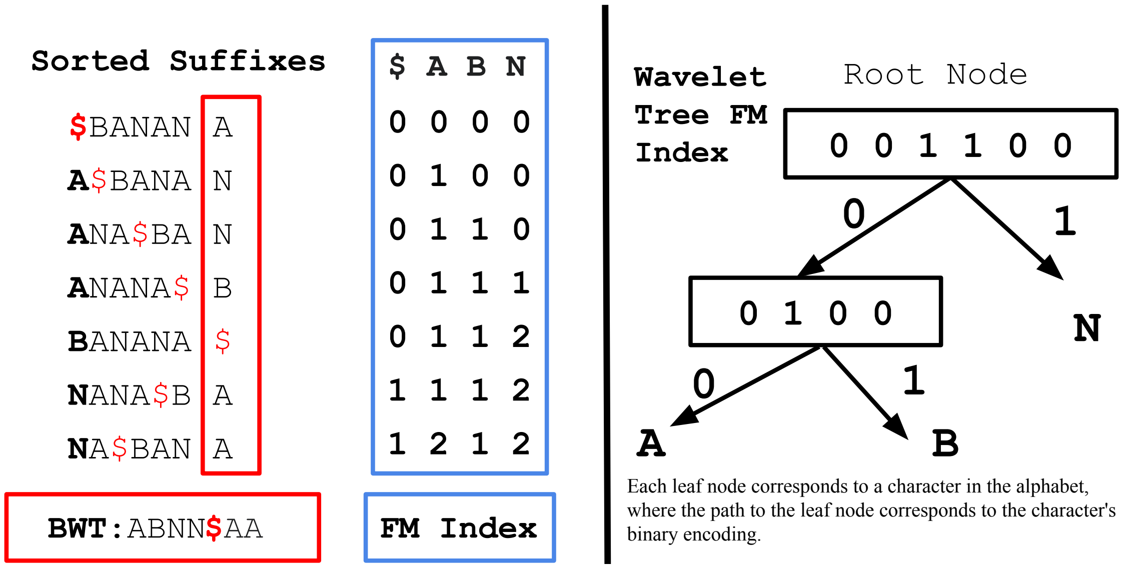


Figure 5.2: Summary of BWT and FM-index on the input string BANANA. For a more illustrated reference see [136]. We show a simple FM Index and a wavelet tree FM-index. To compute $rank(B, 4)$ with a wavelet tree, we first lookup B's binary representation 01. Since the first digit is 0, we find $rank(0, 4) = 2$ in the bitvector at the root node. Then we go down the left branch and find $rank(1, 2) = 1$ as the result.

to ensure comprehensive results, particularly when the log management framework's tokenization scheme is unfamiliar and missing matches is unacceptable.

In the authors' experience operating large-scale distributed systems in industry, substring queries dominate incident response workloads to debug failures and detect intrusions. Efficient support for substring queries is thus a basic requirement for LogCloud.

5.2.3 Solution: The BWT and FM-index

What object-storage based secondary index would allow efficient substring searches on the term dictionary? Apart from FSTs and SSTables, two other full-text indexing approaches have been proposed in literature. The first are grammar-based compression approaches like Sequitur [38, 44, 106, 154, 155] and the second are succinct data structures like the Burrows Wheeler Transform (BWT) [12,41,61,92]. We choose the second approach in LogCloud for two reasons. First, grammar-based approaches heavily rely on repeated subwords that occur frequently in natural language text but rarely occurs in the URIs that we are indexing. Second, while the compression costs of Sequitur-based algorithms can be prohibitively high, efficient industrial-grade implementations for performing the BWT exist [92, 102, 154].

LogCloud uses the FM-index based on the BWT, an example of which is shown in Figure 5.2. The FM-index is a common data structure typically used in bioinformatics to perform substring searches in DNA read mapping. While we will attempt to give a brief tutorial of the BWT in this chapter, the reader is recommended to read the amazing post here: https://curiouscoding.nl/posts/bwt/, or watch the BWT tutorial by Prof. Langmead on Youtube.

To obtain the BWT of an input text corpus (the term dictionary in our case), generate a matrix of cyclic permutations of the corpus, i.e. all the rotations of BANANA in the example. Then, these permutations are sorted lexicographically. The last column from the array of suffixes, highlighted in the red box, is called the BWT [60].

Algorithm 3 Iterative Substring Search using FM-index with BWT

```
procedure FM_Search($P, BWT$)                          ▷ $P$ is the substring to search
    $l \leftarrow 0$, $r \leftarrow |BWT|$
    $C \leftarrow$ counts of each character in $BWT$
    for $i \leftarrow |P|$ down to 1 do
        $l \leftarrow C[P[i]] + rank(P[i], l)$
        $r \leftarrow C[P[i]] + rank(P[i], r)$
        if $l \geq r$ then return "Pattern not found"
        end if
    end for
    return Pattern found between BWT positions $l$ and $r$
end procedure=0
```

The BWT is used to construct the FM-index, which allows efficient substring searches. The FM-index enables efficient computation of $rank(c, i)$, defined as *how many times character c has appeared up to position i in the BWT*. Assuming $rank(c, i)$ can be efficiently computed for all characters in the alphabet and all positions in the BWT, Algorithm 3 is commonly used to find all occurrences of a substring P in the input text corpus using the rank operation repeatedly [61]. Figure 5.2 shows the simplest FM-index, which just records this number for all c and all positions.

The FM-index is typically implemented with a *wavelet tree* in RAM or disk-based used cases [69, 84, 96]. The wavelet tree compresses the BWT into a binary tree, where each node contains a bitvector. To retrieve the rank of a character, the tree is traversed from the root with rank operations done on the bitvectors at each node. A tree traversal for the BWT "ABNNAA" is in Figure 5.2.

The result of Algorithm 3 indicates the query pattern is found between positions l and r of the BWT, which needs to be mapped back to locations in the original text corpus. This can be done very quickly with a list that records the offset in the original corpus that corresponds to each position in the BWT, called the *suffix array*. However, this is a list of integers as long as the original text corpus, and is in general very poorly compressible. A common technique used in literature is the *sampled suffix array*, which stores only offsets for every K positions. If a position i's offset is not stored, the FM-index has to be repeatedly consulted to relate position i's offset to $i-1$'s offset until a sampled location is hit [12, 61].

5.2.4 Challenge: Query Latency on Object Storage

To the best of our knowledge, all existing implementations of the FM-index have targeted disk or in-memory scenarios. This is because the FM-index is typically used to map short reads against a reference genome, which rarely exceeds several GBs in size. However, in our scenario, we would like the index to reside on object storage, which has a higher read latency of tens of milliseconds [55]. This raises two critical challenges for the standard wavelet tree FM-index implementation.

The first challenge is the latency of substring search with the wavelet tree. In a wavelet tree, each rank operation takes $O(H_C)$ sequential random reads, where H_C denotes the entropy of the alphabet. Since we are constructing the index on pseudorandom variables like URIs, the entropy is the log of the size of the alphabet. Thus for alphanumeric variables, around six sequential reads to object storage are required to compute one rank operation with the wavelet tree. Algorithm 3 shows that we compute $|P|$ rank operations sequentially. Long queries such as ‘nginx-554b9c67f9-c5cv4‘ can require tens of rank operations, which translates to hundreds of sequential read requests to the object storage.

The second challenge relates to the latency of accessing the sampled suffix array used to map BWT positions back to locations in the input text. While accessing the FM-index up to K times for each mapped BWT position can be acceptable when the FM-index is in memory or on disk, it incurs unacceptable latency for object storage. This is particularly problematic as hundreds of positions potentially have to be mapped. Even though querying each position can be parallelized, making thousands of small concurrent requests to object storage may run into S3 request throttling [16]. Alternatively, one could opt to store the full suffix array, but it has a very high storage footprint, which would annul the benefits we obtain from log compression [61].

5.3 Object Store Native Inverted Index

LogCloud effectively tackles the two challenges by focusing on the **IO-bound** and **latency-bound** nature of object storage, where data retrieval is significantly more expensive compared to processing the data and retrieving 1 byte and 1 MB have similar latency.

Based on these observations, we introduce two key innovations: (1) a custom object storage-optimized FM-index that reduces sequential requests for substring queries from $O(H_C|P|)$ to $O(|P|)$, and (2) a range-reduced full suffix array approach that maintains performance while drastically reducing storage requirements through effective compression. Together, these innovations adapt the FM-index and suffix array to address the challenges of efficient log search on object storage.

5.3.1 Fast Search with Custom FM-Index

We tackle the first challenge through a novel object-storage-optimized implementation of the FM-index, reducing the sequential requests for a substring query of length P from $O(H_C|P|)$

to $O(|P|)$ versus the standard wavelet tree implementation. The BWT is divided into fixed-size chunks, and we compress each chunk and store the rank of every character in the BWT up to the beginning of the chunk in each chunk. The details are in Algorithm 4. The built chunks can be stored contiguously on object storage together with an offsets array that contains the byte range of each chunk. Algorithm 5 can then be used to compute $rank(c, i)$.

Algorithm 4 Build Chunks for Custom FM Index

```
1: function BuildChunks($BWT, chunk\_size(cs) = 4M$)
2:     $chunks \leftarrow []$, $ranks \leftarrow \{c : 0 \text{ for } c \in \Sigma\}$
3:     for $i \leftarrow 0$ to $\lceil |BWT|/cs \rceil - 1$ do
4:         $chunk \leftarrow (compress(BWT[i \cdot cs : (i+1) \cdot cs]), ranks)$
5:         $ranks[c] \leftarrow ranks[c] + count(c, BWT[i \cdot cs : (i+1) \cdot cs])$ for $c \in \Sigma$ ▷ Update global ranks with counts in this chunk.
6:         $chunks.append(chunk)$
7:     end for
8:     return $chunks$
9: end function
```

Algorithm 5 Rank Computation using Custom FM Index

```
1: function Rank($c, i$, chunks, chunk_size (cs))
2:     $chunk \leftarrow chunks[\lfloor i/cs \rfloor]$                ▷ Locate chunk containing pos i
3:     $text, ranks \leftarrow chunk$   ▷ Chunk contains compressed BWT and the ranks of each character up to the start of the chunk
4:     $local\_pos \leftarrow i \bmod cs$
5:     $decompressed \leftarrow$ decompress($text$)
6:     $local\_count \leftarrow 0$                ▷ Compute rank of $c$ in this chunk.
7:     for $j \leftarrow 0$ to $local\_pos$ do
8:         if $decompressed[j] = c$ then
9:             $local\_count \leftarrow local\_count + 1$
10:        end if
11:    end for
12:    return $ranks[c] + local\_count$       ▷ Final rank = rank up to this chunk + local rank.
13: end function
```

This approach requires reading just one chunk to compute the rank and is much simpler than the wavelet tree design. This implementation, inspired by the original FM-index implementation based on the occurrences matrix and Jacobson's rank [60, 76], is not popular for typical disk/RAM-based FM-index implementations because the rank calculation within the chunk is now done on characters, which is much more compute-intensive than rank calculations on bits that have hardware acceleration like popcnt instructions. However, in our IO-bound scenario this computation cost is easily eclipsed by the read cost.

Another reason why this approach is not typically preferred is because uncompressed, it takes around the same space as the input corpus. The wavelet tree representation comes with native

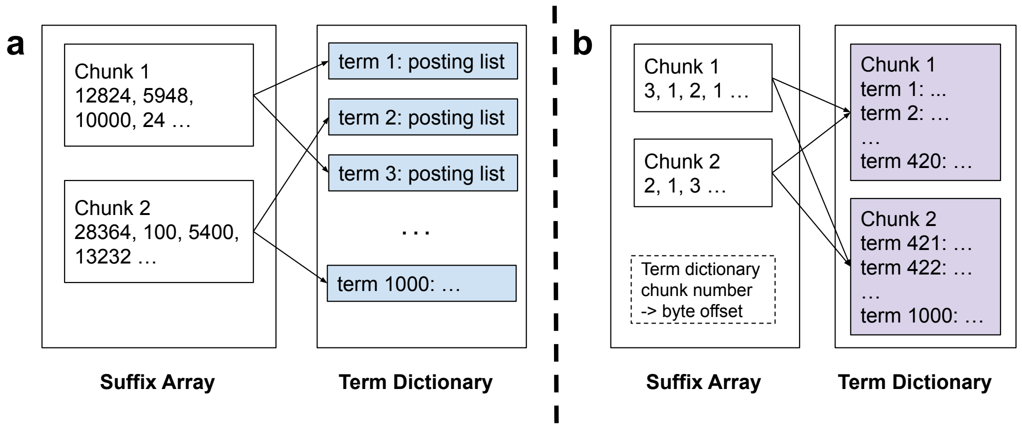


Figure 5.3: Range reduction to compress the suffix array.

compression as the storage footprint of each character is the size of its binary encoding (e.g. Huffman code). However, we can compress each character chunk in our FM-index using generic compression like Zstd [58] and decompress the chunk upon reading. Decompression adds too much overhead for disk/RAM-based FM-indices since reading is fast, but again acceptable in our IO-bound case: decompressing a chunk in memory is much faster than downloading the chunk from object storage. For example, downloading 512 300KB Zstd compressed chunks from S3 in parallel is only 5% slower than downloading and decompressing them concurrently, compared to 70% slower from NVMe SSD on an r6id.2xlarge instance on AWS.

5.3.2 Full Suffix Array with Range Reduction

We resolve the second challenge by storing a heavily compressed full suffix array instead of a sampled suffix array. The FM-index points us to positions l and r in the BWT. We rely on the suffix array to map these positions back to offsets in the term dictionary. As discussed in Section 2.3, the suffix array contains as many 64 bit integers as characters in the term dictionary, whose massive size can negate any of our log compression benefits.

Similar to the FM-index, we store the full suffix array in chunks, and compress each chunk. To fetch positions l to r, the chunks containing those positions are downloaded and filtered for these positions. However, if the chunks contain byte offsets of the posting lists in the term dictionary (Figure 5.3a), they are still very poorly compressible because they would contain a wide range of large integers with minimal patterns or repetition, making standard compression algorithms like Zstd ineffective at reducing their size.

In LogCloud, instead of storing offsets into the original term dictionary, we break the term dictionary into chunks, and only record the chunk number in the suffix array (Figure 5.3b). Even though we still have to store the same number of integers as the naive approach, the dynamic range

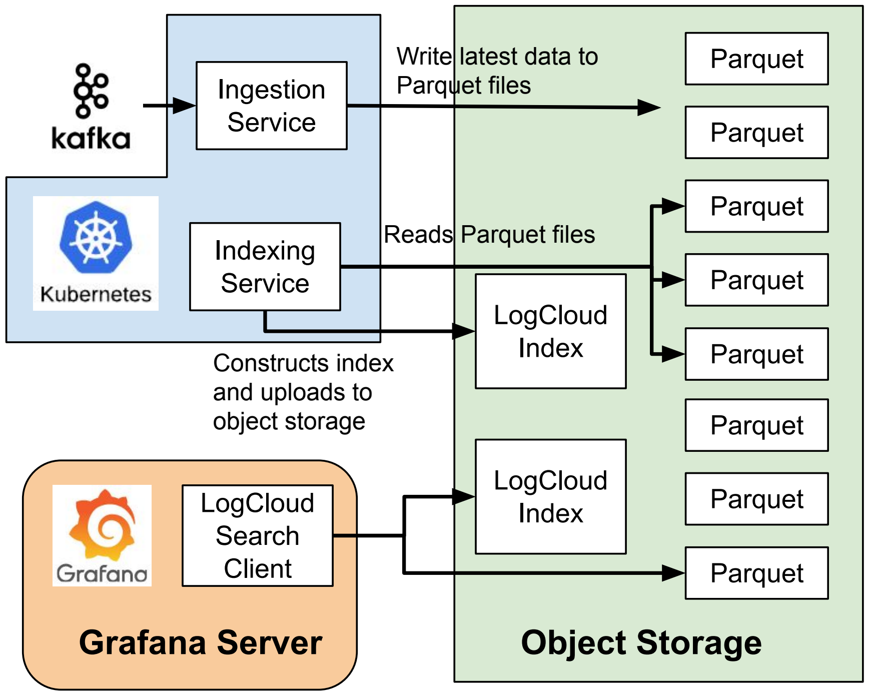


Figure 5.4: LogCloud's architecture.

of each integer is reduced by several orders of magnitude. Subsequent positions in the suffix array are also now more likely to be identical. This makes generic compression like Zstd very effective on the suffix array. We call this optimization technique ***range reduction***.

This optimization is motivated by the observation that byte-range GET requests on object storage up to around 1MB are all latency bound and have roughly the same speed. As a result, it is unnecessary for the secondary index to point us to the exact 20-byte term in the term dictionary. It is sufficient to point to the 1MB chunk that contains the term, then download and scan the chunk exhaustively to locate the term. The scan cost is usually insignificant compared to the download.

5.4 LogCloud Architecture

We now discuss how the novel object-storage native inverted index fits in the overall architecture of LogCloud. LogCloud consists of two key components, indexing and querying, shown in Figure 5.4. LogCloud is implemented as a Rottnest index. In the previous chapter, we explained how Rottnest provides an API for clients to keep an index up to date and query indices. Here, we show how a Rottnest index can be run in production as part of a system.

5.4.1 Indexing

Similar to other compute-storage decoupled log management systems, LogCloud runs an ingestion pipeline that dumps logs in Parquet format on object storage. What sets LogCloud apart from other such systems is that it also runs an indexing service. The indexing service periodically runs the Rottnest 'index' API described in Chapter 4. Once a configurable amount of new logs have been collected, a LogCloud inverted index is built on the new data. By default the data is converted to Parquet and stored on object storage as a directory of Parquet files. Since LogCloud is implemented as a Rottnest index, it can also directly index log data already stored in data lakes from another ingestion service [19, 47, 123].

During indexing, we first use LogGrep [147] to break down ingested logs into template and variable components, categorizing variables into 64 *types* based on their character composition (e.g. only numeric, alphanumeric etc) [147]. For each type, LogCloud builds an inverted index with a term dictionary divided into 1MB chunks, as described in Section 3.2. The posting list points to Parquet pages, which are chunks of a few hundred KBs of compressed data. We find that we can download and search hundreds of Parquet pages in parallel in hundreds of milliseconds from an EC2 instance with a heavily optimized custom Parquet reader in Rust.

If the compressed term dictionary exceeds 5MB, we construct the secondary FM index and suffix array described in Section 3.1 to efficiently look up term dictionary chunk numbers from substring queries. Otherwise, we simply scan the term dictionary. Altogether, a LogCloud index file contains the templates, term dictionary chunks, and optionally the FM index and suffix array.

We can avoid storing a copy of the logs by piecing together matching logs from their templates and variables as in LogGrep, we found this approach requires too many random accesses when the variables are stored in object storage, making search prohibitively expensive. In addition, the Parquet files allow us to easily augment LogCloud with tools like SparkSQL and AWS Athena for more complex analytics.

5.4.2 Searching

One can use Rottnest's embedded client library to search a LogCloud index and Parquet files on object storage. This offers flexible deployment options anywhere that can access the object storage bucket containing these files, such as on a Grafana server or on a serverless function. The search functionality operates completely independently from indexing and requires no always-on servers.

The search process for a top-K substring query is illustrated in Figure 5.5. The latest unindexed data is scanned in Parquet directly. LogCloud queries all the built LogCloud indices in parallel. To query each index, the following steps occur:

- The extracted templates are downloaded from object storage and searched exhaustively. If the substring query matches here, the searcher will simply abort using the index and brute force

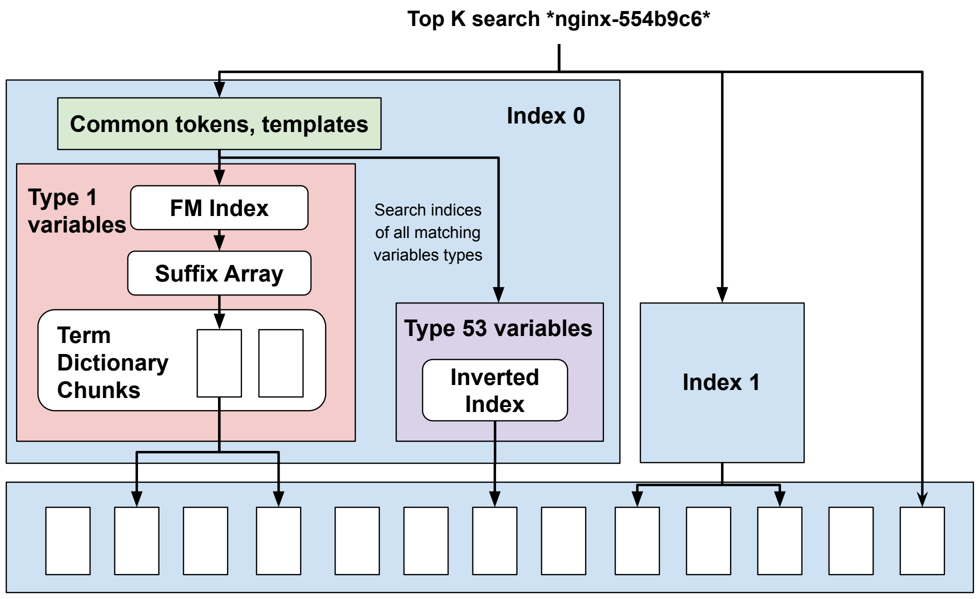


Figure 5.5: Searching workflow in LogCloud. All data structures shown in boxes are stored on object storage.

search all the Parquet files since the target substring occurs frequently.

- URI substring searches, e.g. "$*55493*$", will not match common templates, leading the searcher to search the inverted indices for the variables, described in Section 5.3. The LogGrep type of the query is determined and the inverted indices for all "compatible" types are searched in parallel. A compatible type is a type that could contain the type of the query. For example, if the query contains only numbers (type 1 in LogGrep), the type containing all alphanumerics also must be searched (type 53).[2]

- The search client first queries our custom FM-index to find positions l and r in the suffix array as described in Algorithm 1, then retrieves term dictionary chunk numbers from the suffix array between these positions. The chunks are downloaded and regex searched for matches, with any matches leading to retrieval and search of the referenced Parquet file pages.

Certain parts of the index, like the templates and FM-index metadata, are small and accessed repeatedly across queries. While these characteristics make them ideal candidates for client-side disk caching, we do not explore this in our evaluation to maintain a straightforward comparison with other systems due to the high variability in cache-hit rates across various log analytics use cases.

[2]A current limitation is LogCloud cannot search queries that span templates and URIs. This is exceedingly rare in practice.

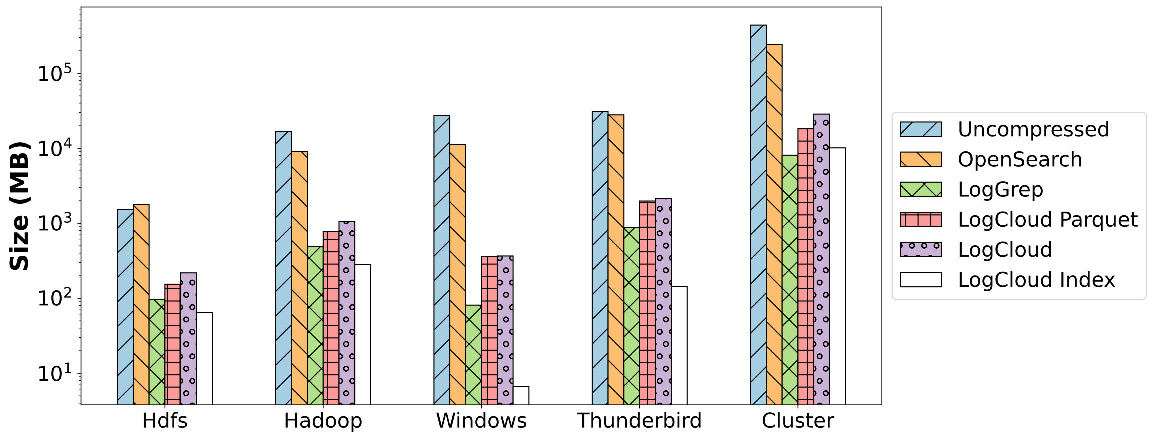


Figure 5.6: Storage footprint comparisons across five datasets.

5.5 Results

We compare LogCloud against two representative baseline systems: OpenSearch UltraWarm, which exemplifies compute-storage integrated indices, and LogGrep, which represents the compute-storage disaggregated approach of downloading and scanning compressed logs [18, 147]. For the LogGrep baseline, we compress the logs using LogGrep and store them in object storage. During search, the compressed logs are downloaded and searched on NVMe SSD.

We use four LogHub datasets, HDFS (1.5GB), Thunderbird (30GB), Hadoop (17GB), Windows (26GB) [147, 157], as well as a 429GB dataset named Cluster from [118]. For each dataset, we test three search queries: common keyword, exact-match URI, and substring URI, returning top 1000 results. For example, on the Hadoop dataset, we search for 'blk_1076115144∗', '∗1076115144∗' and ERROR.

We run LogCloud and LogGrep searcher on a single r6i.xlarge EC2 instance (4 vCPUs, 32GB RAM) [137], with LogCloud indices, Parquet files and LogGrep compressed files stored on AWS S3 in the same region. AWS OpenSearch UltraWarm uses three r7g.large nodes (2vCPU, 16GB RAM) and three ultrawarm1.medium instances (2 vCPUs, 15.25GB RAM). All measurements are repeated five times, with the standard deviation shown where applicable.

5.5.1 Storage Footprint

First, we compare the storage footprint of the different log management solutions in Figure 5.6. For LogCloud we show both the size of just the Parquet files and the total size with the index files. Across all five datasets, LogCloud (Parquet + index) achieves 11.8x geomean lower storage footprint against OpenSearch and 2.8x larger storage footprint compared to LogGrep. The LogCloud index itself achieves 93x geomean lower storage footprint compared to OpenSearch and 2.8x lower compared to LogGrep. We make the following observations:

[leftmargin=*]Consistent with prior findings [118, 147], OpenSearch exhibits poor space efficiency, with its index size approaching that of the raw uncompressed logs. Moreover, OpenSearch UltraWarm incurs additional operational costs due to its requirement for continuously running servers, described more in Section 5.5.3. For most log types, LogCloud storage size is dominated by the Parquet files. As expected, LogGrep's storage footprint is smaller since its log-specific compression outperforms the Zstd compression used in Parquet [147].

The second observation raises the question if we can further improve the storage footprint by moving away from Parquet: instead of Parquet's zstd compression, we could use LogGrep to compress chunks of logs and have LogCloud's posting lists point to those chunks. However, this would sacrifice crucial interoperability with external SQL engines and data lakes [27, 62, 123].

5.5.2 Search Latency

In Figure 5.7 we show the search performance of the three different query types on the five log datasets with OpenSearch, LogGrep and LogCloud. We break down the LogGrep runtime into the time it takes to download the compressed logs and the time to search the downloaded files on disk. We break down the LogCloud runtime into searching the index on object storage and downloading and filtering the matched Parquet pages.

On the queries on common tokens such as "ERROR" that match log templates, LogCloud sidesteps the inverted index and directly searches Parquet files, spending almost no time in index search. This leads to a 2.2x geomean speedup over OpenSearch and 5.0x over LogGrep, consistent over almost all log types.

For URI queries, LogCloud's index search dominates query time over Parquet page retrieval, as expected for an effective inverted index. OpenSearch UltraWarm's FST-based secondary index outperforms LogCloud on exact matches (2.5x geomean faster on average), except for Windows. However, LogCloud is 3.6x faster on substring queries which bypass OpenSearch's FST index, achieving up to 7.5x speedup on the largest Cluster dataset. Most notably, LogCloud matches this performance using only S3 storage instead of OpenSearch's disk/RAM-based indices, demonstrating competitive serverless query performance without the costs of warm storage.

On the small dataset Hdfs, LogGrep performs better than LogCloud since the compressed logs can be quickly downloaded and scanned. However, it performs over 10x worse on the larger datasets like Hadoop and Cluster, where the search time on disk actually eclipses download time. Since LogGrep does not rely on indices, it has to exhaustively scan all the variables, causing its poor scalability. As a result, on URI queries, LogCloud is 22x geomean faster than LogGrep for Hadoop and 12x for Cluster.

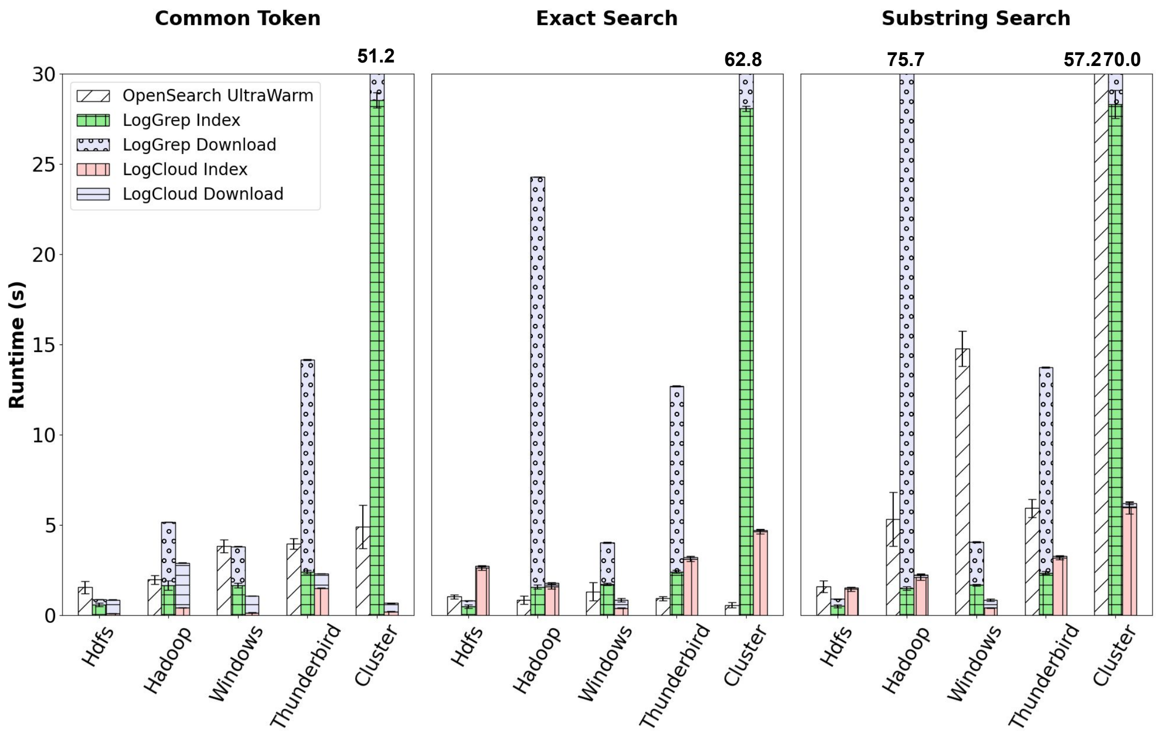


Figure 5.7: Search times for different query types across five public log datasets with breakdowns between search and download for LogGrep and index and Parquet for LogCloud. Bars exceeding the y-axis are annotated.

5.5.3 Total Cost of Ownership

Our analysis so far across log volumes ranging from 1.5GB to 429GB demonstrates that LogCloud consistently maintains a middle ground between LogGrep and OpenSearch in terms of storage footprint and search latency. We now examine how these performance characteristics, combined with indexing costs, influence the total cost of ownership (TCO) of the log management system.

The precise question we seek to answer is given a log dataset and a fixed operating horizon, say 12 months, what is the most cost effective system for a particular total query load? To answer this question, we estimate the TCO of OpenSearch, LogGrep and LogCloud as follows:

- **OpenSearch UltraWarm** is typically operated as a longrunning cluster, where the cost consists of many components such as searcher nodes, UltraWarm nodes, EBS cost and S3 cost for UltraWarm. While the operating cost of the smallest OpenSearch cluster could easily exceed \$1500/month [18], we adopt a strict lower bound of this TCO here, which is just the storage cost of the primary shard of the index in S3 and the smallest required UltraWarm node (\$174/month), ignoring ingestion cost.
- **LogGrep**'s cost can be estimated as: compression cost (compression time $\times$ cost of EC2 instance) + storage cost (compressed logs size $\times$ S3 cost) + search cost (*representative search latency* $\times$ cost of EC2 searcher instance $\times$ the total number of queries).
- **LogCloud**'s operating cost can be computed with the same components as LogGrep. In this case, the storage footprint contains both the Parquet and the LogCloud index. LogCloud builds more indices on top of LogGrep, having a 10.7x geomean higher indexing cost on all the log types. However, the absolute cost is still quite low, only \$3.6 on the 429GB Cluster dataset and less than \$1 on all other datasets.
- In addition, we add another TCO comparison against **Datadog Flex logs**. We use the Flex "Starter" pricing at \$0.1/GB ingested and \$0.6 per million records per month. Datadog also offers Flex pricing with \$0.05 per million record per month with a fixed compute commitment, though the smallest such commitment would exceed the cost of all other systems considered here [132]. We also considered Grafana Loki [68], though we found LogGrep to be more economical in all cases.

We estimate the representative search latency for LogGrep and LogCloud here by averaging the latencies of different types of queries in our benchmark. Based on this estimate of the representative search latency, we can then compute the TCO at different query loads and plot the phase diagrams as explained in Chapter 4, shown in Figure 5.8.

In Figure 5.9, we plot LogCloud's TCO saving over the next best approach at different query loads for the different log types with a 12 months operating horizon. This amounts to drawing a vertical line on the phase diagram at 12 months, but in addition shows the quantitative TCO wins

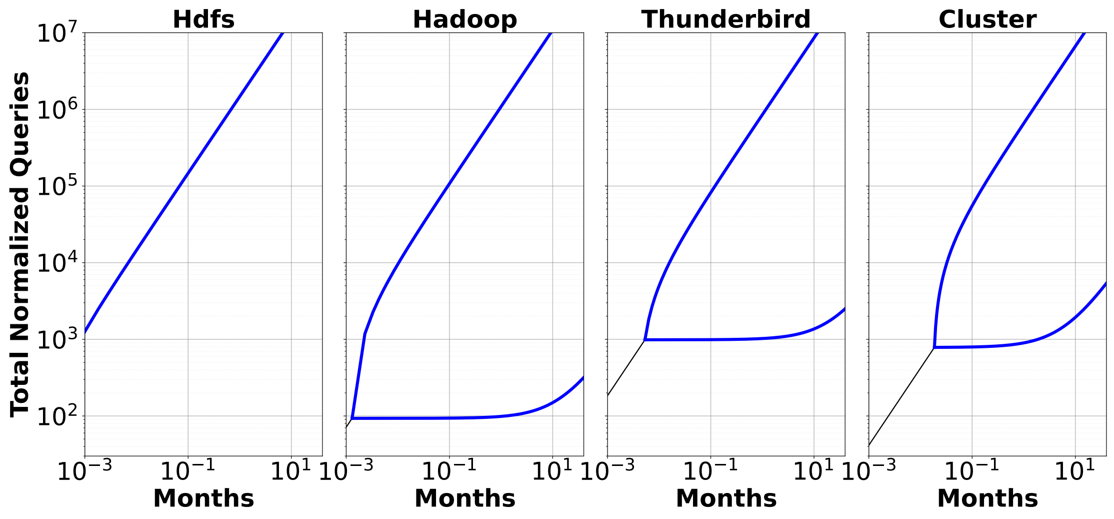


Figure 5.8: Phase diagrams for the four different log types.

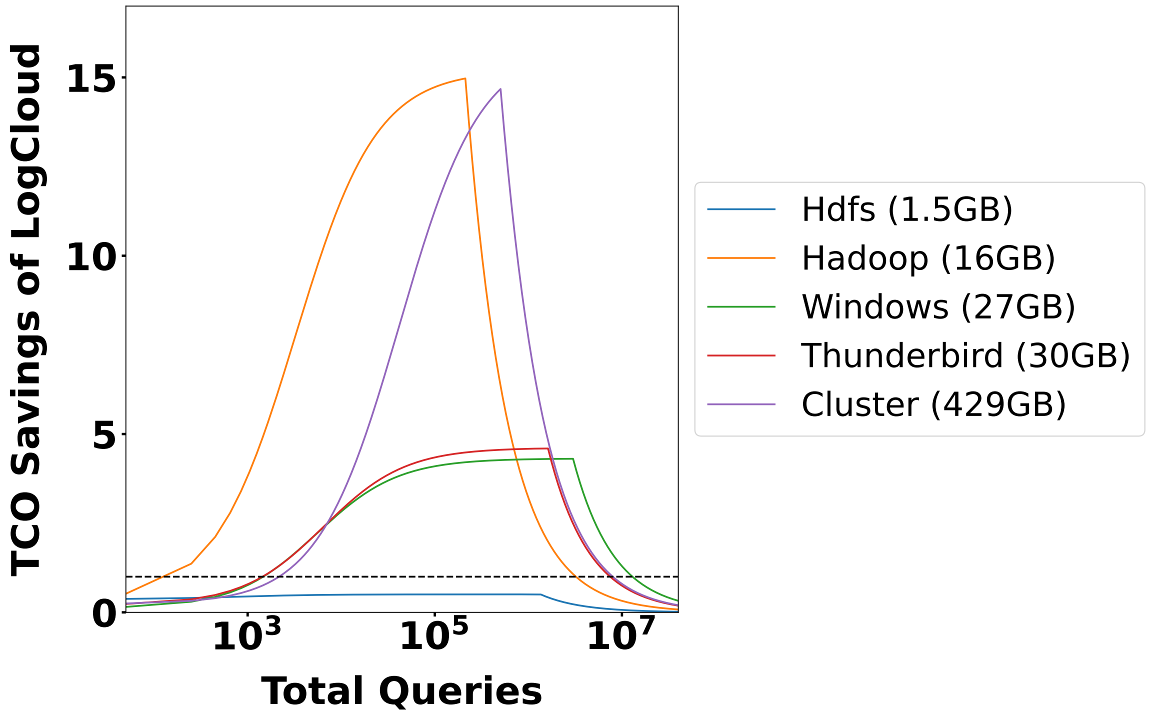


Figure 5.9: LogCloud's TCO savings compared to the cheaper of LogGrep and OpenSearch Ultra-Warm at 12 months. Each solid curve corresponds to a different log type. The line at 1 represents where LogCloud outperforms baselines.

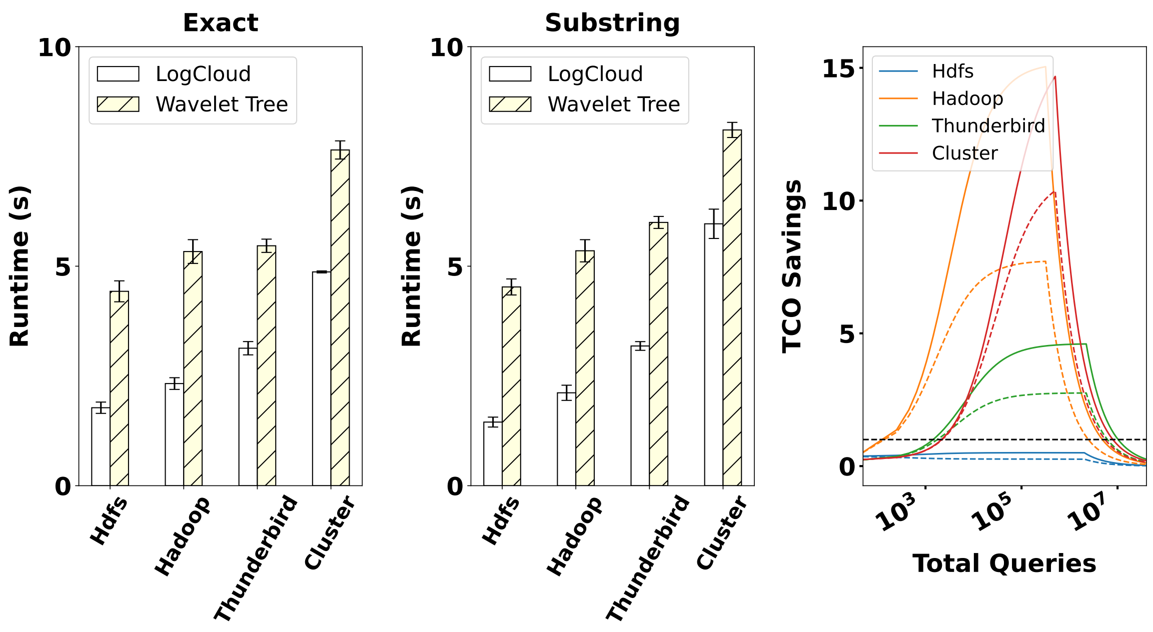


Figure 5.10: LogCloud index search times for exact and substring queries with custom FM-index vs wavelet tree. The solid line denotes the TCO profile of the custom FM-index, whereas the dashed line indicates that of the baseline wavelet tree.

of LogCloud over competing methods. The plot exhibits a distinctive peak shape. OpenSearch UltraWarm and Datadog have free search cost but high storage cost, LogGrep is the opposite, while LogCloud is in between the two approaches. As a result, at very low query loads LogGrep is more cost efficient whereas at high query loads OpenSearch is more efficient. The peak occurs where OpenSearch or Datadog surpasses LogGrep in terms of cost efficiency: we find OpenSearch to be more cost efficient than Datadog for Thunderbird and Cluster, with Datadog better on the other three datasets. These systems surpass LogCloud in cost efficiency at around 10^7 queries.

For large-scale datasets like Hadoop and Cluster, LogCloud achieves optimal cost-efficiency in a ”sweet spot” query volume range spanning around four magnitude from 1000 to 10^7 total queries, where it can be up to 15x cost-effective than alternatives. Importantly, we note that the total range of queries where LogCloud wins is consistent across the different log types, enabling practitioners to reliably predict its effectiveness for new log sources.

5.5.4 Ablation Studies

LogCloud’s main technical novelty lies in the custom FM-index described in Section 5.3. In Figure 5.10, we show LogCloud’s index search time with an optimized wavelet tree implementation [61,70]. The subsequent Parquet access speed is not compared as it is the same between the two strategies, which download the same pages. We skip this analysis for the Windows dataset as no secondary

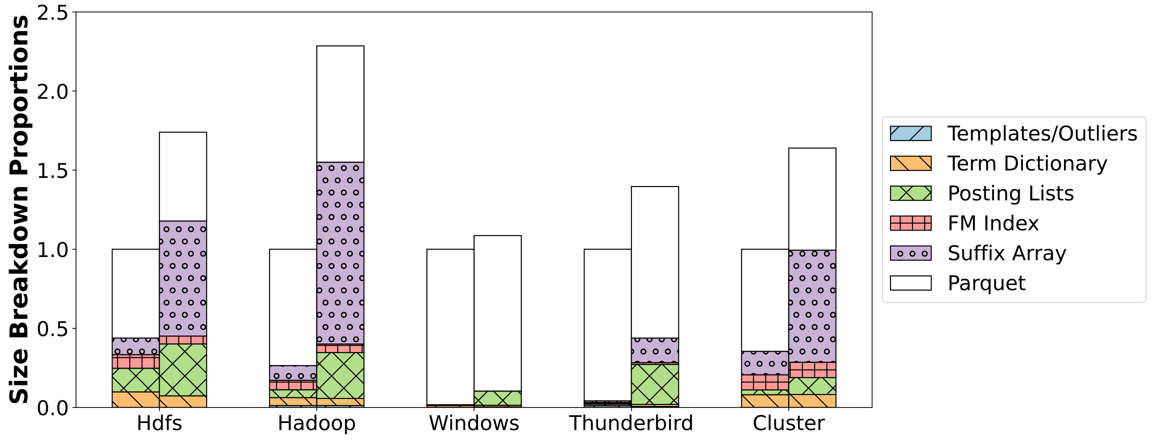


Figure 5.11: LogCloud component sizes with (left bars) and without (right bars) the range reduction optimization.

index was constructed. Across the remaining queries, our custom FM-index achieves a geomean 2.2x speedup over the wavelet tree baseline, significantly increasing LogCloud's TCO advantage.

In Section 5.3.2, we introduce the range reduction optimization that compresses the suffix array by storing term dictionary chunk numbers instead of individual term offsets. In Figure 5.11, we show that for three log types, the suffix array was the largest component before this optimization. The optimization reduces the suffix arrays' size by geomean 8.8x, after which the Parquet files dominate the storage footprint, marking diminishing returns for further index size optimizations. Tuning the term dictionary chunk size provides direct control over the suffix array size - Figure 5.12a shows we can reduce it by nearly 90% (from 286MB to 30MB) simply by adjusting the chunk size from 10KB to 4MB, while the FM-index size remains relatively stable. However, this reduction has a tradeoff: as shown in Figure 5.12b, larger chunks require more exhaustive scanning during searches, increasing substring query times from 1.7s to 3s.

5.5.5 Scalability

Our evaluation shows that LogCloud maintains both low storage footprint and search latency scaling to datasets up to 429GB, achieving up to 10x TCO savings by avoiding OpenSearch and LogGrep's poor scalability at high scale due to the lack of appropriate indexing support. In this section, we control for the variability in log content and examine LogCloud's scaling along two axes, dataset size and operating horizon, by examining just the Cluster dataset.

In Figure 5.13a, we plot the TCO savings curve for the Cluster dataset at different subsampled sizes. We see that the cost benefits of LogCloud increases significantly at larger dataset sizes over a stable range of total queries of around four orders of magnitude, confirming LogCloud's advantage at higher scales over OpenSearch UltraWarm and brute-force scanning with LogGrep.

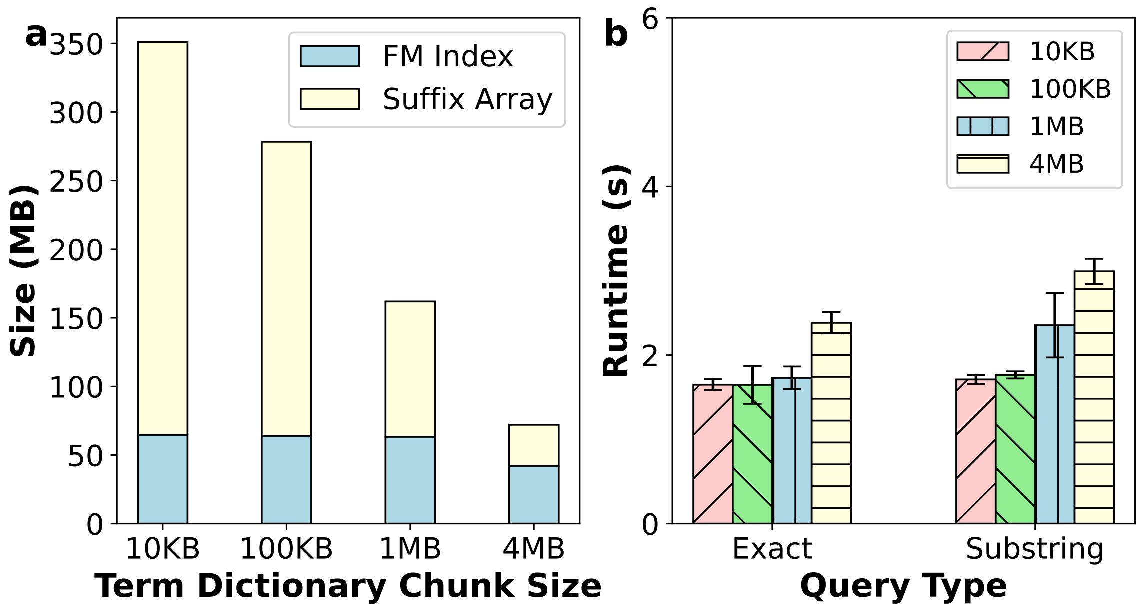


Figure 5.12: a) Term dictionary target chunk size vs index size. b) Search latency by type vs. term dictionary chunk size.

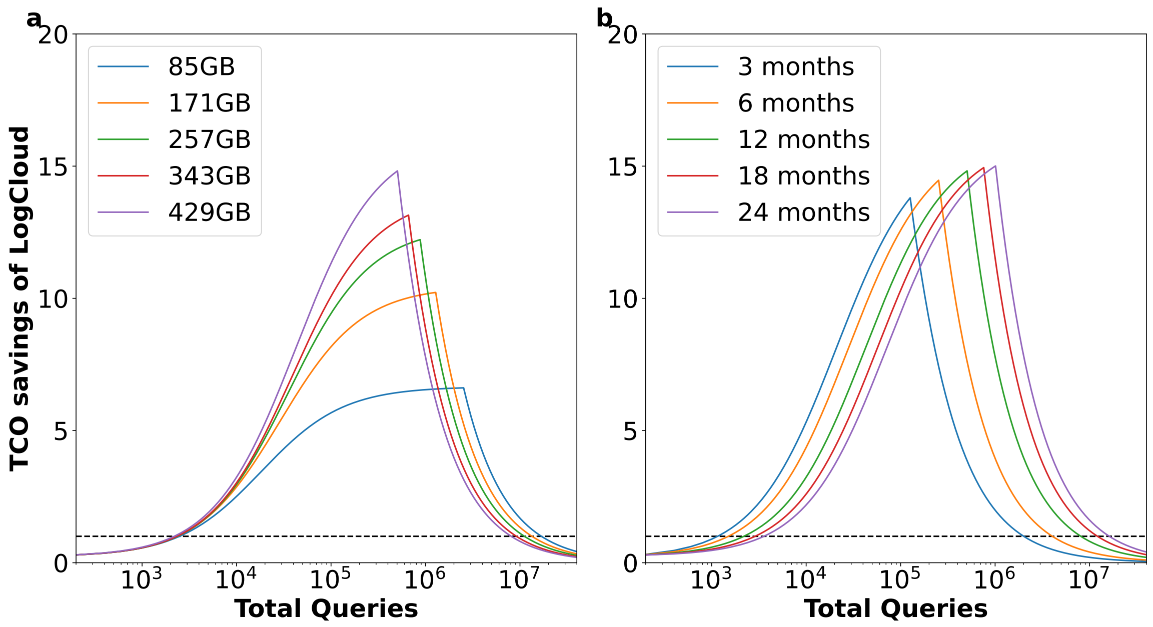


Figure 5.13: How the TCO ratio curve for Cluster shifts at a) different dataset scales and b) different operating horizons.

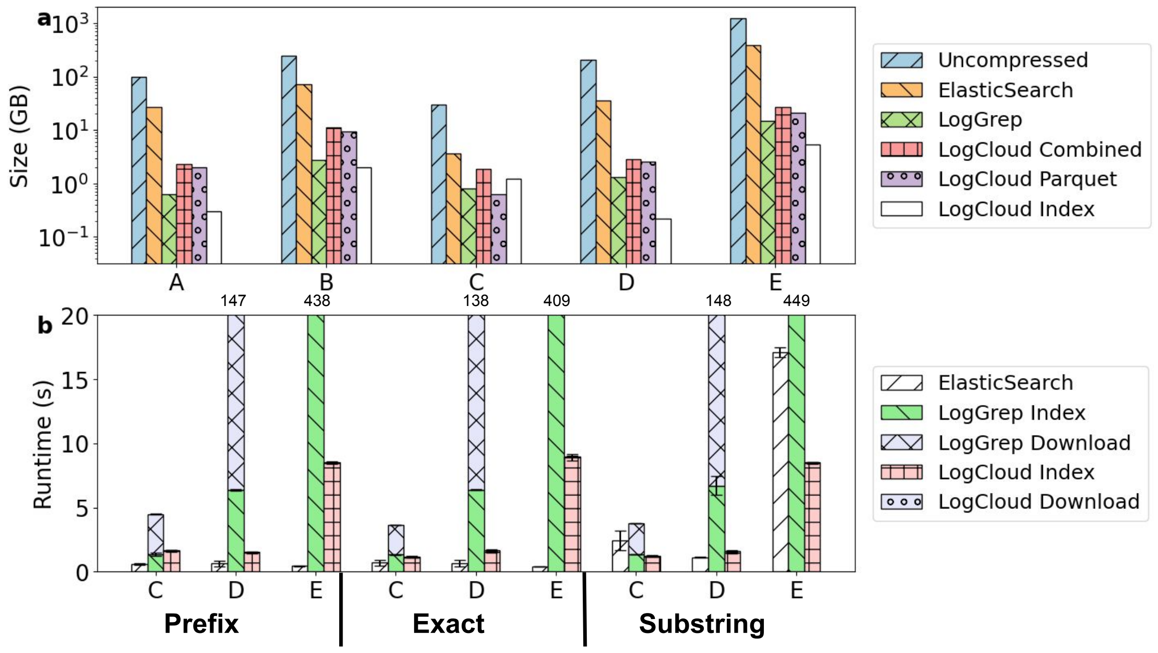


Figure 5.14: a) Storage footprint and b) Search times of ElasticSearch, LogGrep and LogCloud.

Figure 5.13b shows how operating horizons affect LogCloud's TCO curve. LogCloud's cost advantages increase with longer durations before converging. With longer operating times, LogCloud becomes cost-effective at higher query volumes but maintains advantages at high loads. This shift is beneficial since longer operations typically involve higher query volumes.

5.5.6 Production Test Case

We also tested LogCloud on the real production logs generated by a hosted service of a major public cloud provider, with the biggest around 1.2TB in size. The logs are produced in Json format and currently stored and queried in self-hosted ElasticSearch hot-tier with a set retention period. We tested LogGrep and LogCloud using the same configurations as the open source datasets.

Figure 5.14 shows that similar to the public log datasets, LogCloud significantly reduces storage footprint compared to ElasticSearch, achieving geomean 8x on the five datasets. We also show the performance of nine URI substring queries on log types C, D and E. As expected, LogGrep performs acceptably on the smaller dataset C, but is much worse than the other two options on the larger datasets D and E. In contrast to OpenSearch UltraWarm used on the public log datasets, the ElasticSearch hot-tier service stores the index entirely in memory/SSD, leading to even lower search latencies for prefix and exact URI queries. However, it is still significantly slower for substring queries, particularly for very large datasets such as E. For these queries, we see LogCloud's pure object-storage based design can still outperform ElasticSearch by geomean 2x.

5.6 Conclusion

In this chapter, we show how Rottnest can be leveraged to provide a compelling log management system for observability applications. We envision LogCloud to be used alongside existing log management solutions like ElasticSearch, Splunk or DataDog in practice [50, 57, 129]. Near-line logs can be offloaded to LogCloud to reduce cost while critical online log analytics remain on the existing solutions. Future work could include an efficient pipe-based query language like Splunk's SPL and better integration with other query engines like Trino.

Chapter 6

Conclusion

Now I will talk about what I have learned about distributed systems theory and practice in my PhD.

6.1 The Impermanence of Forms and The Eternity of Essence

Distributed systems face the formidable challenge of managing terabytes of state – from shuffle batches flowing through compute clusters to vast tabular datasets in data lakes. The key to tractable state management lies in a powerful abstraction: rather than persisting massive data volumes directly, systems maintain compact metadata that serves as both a succinct representation and a replayable log of system state. This approach exploits a fundamental duality – while the actual data remains logically immutable (ensuring deterministic recovery), its physical manifestation can be entirely ephemeral, allowing systems to dynamically optimize where and how data is stored, cached, or regenerated based on current resource availability and performance requirements.

6.1.1 The Power of State Tracking

State tracking in distributed systems enables transformative capabilities. This dissertation demonstrates through multiple case studies how careful state management unlocks features that would otherwise be impossible or prohibitively expensive. Quokka leverages state tracking to enable fault recovery – when failures occur, the system can "rewind" to a consistent state and resume processing without losing correctness. Rottnest uses historical state snapshots to perform incremental indexing, computing only what changed between two points in time rather than reprocessing entire datasets. These capabilities fundamentally depend on the system's ability to capture and reason about its state over time.

6.1.2 The Fundamental Challenge: Data Volume

The core challenge in state management stems from the sheer volume of data that distributed systems process. Consider the scale: in Quokka, the "state" includes all data batches flowing through every operator across the entire cluster. In Rottnest and other data lake systems, it encompasses every Parquet file or index file stored on object storage. Naively persisting all this data for fault tolerance or versioning would overwhelm any storage system and make recovery impossibly slow.

This volume challenge forces us to think differently about state representation. We cannot afford to checkpoint GBs or TBs of raw data continuously. Instead, we need a more sophisticated approach that succinctly captures the essence of the system state.

6.1.3 The Key Insight: Metadata as State Through Immutability

The breakthrough that makes practical state management possible is a simple but powerful observation: when data objects are immutable, we can use metadata as a perfect proxy for the data itself. This immutability creates a one-to-one mapping between a piece of metadata and the data it represents.

In Quokka, once a data batch is created with specific lineage information, its contents never change. The lineage metadata – which task created it, with what inputs, at what sequence number – uniquely identifies that batch forever. As long as we persist this lineage *metadata*, the underlying data can always be reconstructed. Similarly, in data lake systems like Iceberg and Delta, Parquet files are immutable. A simple filename suffices to represent megabytes or gigabytes of actual data. It thus suffices to only record what *filenames* a Rottnest index file points to, not the actual *data*.

Even systems like Apache Kafka leverage this principle: once a message is written to a partition at a specific offset, it becomes immutable. Consumers need only track their offset metadata – a single integer per partition – to know exactly what they have and haven't processed. The actual message data need not be duplicated in consumer state.

However, not all systems can leverage this approach. PostgreSQL's Write-Ahead Log (WAL) represents the opposite extreme: it must record actual data changes, not just metadata, because rows in tables are mutable. When a transaction updates a row, PostgreSQL must log the actual before and after values to ensure durability and enable replication. There's no stable metadata proxy for mutable data.

This insight transforms an intractable problem into a manageable one. Instead of persisting massive amounts of data, we persist compact metadata that can reconstruct or reference that data when needed – but only when immutability guarantees exist.

6.1.4 Flexibility Through Logical State Representation

Metadata representation provides another crucial benefit: it separates logical state from physical implementation. When Quokka recovers from a failure, it doesn't need to restore the exact physical state – channels can be scheduled on different machines, and data might be received out of order on the consumer. However, the metadata tells the tasks in those channels exactly how to re-sequence their out-of-order inputs to reproduce their outputs.

This separation is evident across many systems. Kubernetes exemplifies this principle: it persists the *desired state* as YAML specifications in etcd – how many replicas should run, what resources they need, which nodes they prefer. The actual physical state – which pods run on which nodes, their IP addresses, their current resource usage – is ephemeral. Kubernetes continuously reconciles physical reality with logical intent, allowing pods to be rescheduled, nodes to fail, and IPs to change while maintaining the same logical application state.

Similarly, Apache Spark's RDD lineage tracking separates logical computation graphs from physical execution. An RDD's lineage describes what transformations to apply, not where or how to execute them. During recovery, Spark can recompute lost partitions on any available executor.

This separation enables powerful optimizations. During recovery, Quokka can schedule computation on completely different workers based on current cluster availability. Data lakes and Rottnest can compact, sort, or reorganize files between snapshots; as long as queries return the same logical rows, the physical layout is irrelevant. This flexibility is only possible because metadata captures what the state *should be* rather than what it *physically is*.

6.1.5 Logical Immutability Affords Physical Ephemerality

The apparent tension between this physical ephemerality and the immutability principle from the previous section resolves when we recognize they operate at different levels. Immutability applies to logical data objects: once created, a Quokka data batch with specific lineage, a Parquet file with particular contents, or a Kafka message at a given offset never changes. This immutability enables metadata to serve as a reliable proxy.

Physical ephemerality, on the other hand, refers to where and how these immutable objects are stored or computed. It is precisely because objects are *logically* immutable that they can be *physically* ephemeral, because we always know how to reconstruct them. A Parquet file remains logically immutable even as it moves between storage tiers or gets cached on different nodes. A Quokka data batch can be recomputed on any machine because its lineage metadata guarantees it will have identical contents. The key insight is that immutable logical objects can have ephemeral physical manifestations – the metadata serves as the bridge between these two levels, ensuring logical consistency while enabling physical flexibility.

6.1.6 Choosing What to Track

Even with metadata as our representation, critical decisions remain about what information to persist. These decisions involve fundamental trade-offs:

Minimalist approaches persist only the bare minimum required for correctness. Quokka exemplifies this philosophy – its task lineage and sequence numbers form an extremely concise representation. Apache Flink takes a middle ground with its asynchronous checkpoints: it snapshots operator state and in-flight records at coordinated points, but doesn't persist all intermediate data. During recovery, the system must recompute all intermediate data from these sparse breadcrumbs. This approach minimizes storage overhead but sacrifices recovery time.

Comprehensive approaches persist additional auxiliary state to speed recovery. Redis, for instance, offers both RDB snapshots (complete memory dumps) and AOF logs (every write operation). Apache Samza similarly checkpoints all task state to a changelog topic in Kafka, enabling near-instantaneous recovery at the cost of continuous state replication.

The optimal choice depends on system-specific factors: How expensive is recomputation relative to storage? How frequently do failures occur? Are inputs replayable? There's no universal answer—each system must analyze its own requirements and constraints.

6.1.7 Persistence Technologies and Access Patterns

Beyond choosing what to represent, systems must decide how to persist their metadata based on access patterns:

Single-state systems like Kubernetes only need the current desired state. They continuously reconcile actual state with desired state, never needing historical versions. For these systems, a transactional key-value store like etcd provides fast access to the current state with strong consistency guarantees. Apache Mesos and HashiCorp Consul follow similar patterns, maintaining only current cluster state in their respective stores.

Time-travel systems like data lakes and Rottnest need access to historical states. They must support queries like "what data existed at timestamp T" or "what changed between T1 and T2." These systems typically use append-only logs that store state deltas. Git exemplifies this approach at a different scale: it stores content-addressed objects (blobs, trees, commits) immutably and uses references (branches, tags) as mutable pointers into this immutable history.

Hybrid systems might need both patterns, for example when the system requires time-travel (e.g. rollbacks) as well as a materialization of the current state for purposes such as querying. Elasticsearch maintains current cluster state in-memory with disk persistence while also writing an operations log (translog) for durability. PostgreSQL maintains current table state on disk while using WAL for both crash recovery and replication to standby servers.

6.1.8 Design Principles for Practical State Management

This dissertation's case studies reveal three fundamental principles for designing state management in distributed systems:

1. **Identify and persist minimal required state**: Determine what information is absolutely necessary for correctness. This forms the non-negotiable core that must be persisted reliably, whether it's Quokka's lineage information or a data lake's file metadata.
2. **Leverage immutability for compact representation**: Use metadata as a proxy for data wherever possible. This requires careful system design to ensure immutability guarantees, but enables orders-of-magnitude reductions in state size.
3. **Match persistence strategy to access patterns**: Choose storage technologies and granularities based on how state will be accessed. Don't over-engineer – a system that only needs current state shouldn't pay the complexity cost of temporal versioning.

6.2 The Cloud as a New Computing Medium

Cloud is a new compute medium with interesting and evolving properties. Unlike traditional computing environments with fixed resources, the cloud offers elasticity, heterogeneity, and disaggregated storage – but also introduces new challenges like spot instance preemptions and higher storage access latencies. More importantly, the cloud offers a new set of primitives, like persistent key-value stores, cheap and reliable long-term storage and serverless compute, that can provide a basis to build new distributed systems.

6.2.1 The Research Gap in Cloud Engineering

It is my belief that there is a paucity of research in this new field of "cloud engineering". There are major gaps not covered by both industry standard infrastructure-as-code (IaC) tools like Terraform and Pulumi and academic research projects like Hydroflow [120].

While modern IaC tools allow users to easily provision and configure individual resources in AWS, they are typically focussed on spinning up common backend infrastructures for applications. They leave all state management and fault tolerance logic to the user, and are not meant to build new distributed systems like Quokka or Rottnest.

On the other hand, tools like Hydroflow are very low level. While one could conceivably build a new key-value store with such tools, it does not allow users to easily integrate with existing key-value stores like DynamoDB as part of their applications.

6.2.2 Future Directions

Future work in this area could include better DSLs to build distributed systems based on existing cloud primitives (i.e. persisting state snapshots to managed key-value stores like DynamoDB or log entries to object storage like S3), which would greatly simplify the construction of systems like Quokka.

Another interesting direction might be a new protobuf-like storage format specification that formalizes the componentization optimizations for Rottnest indices. An in-memory data structure would be automatically decomposed into components based on expected data access patterns, where different compression algorithms and storage lifecycle management policies could be applied to the different components to target different latency and throughput tradeoffs. For example, in Figure 4.6, the root component in the binary search tree might be automatically optimized to a look up table cached in an in-memory key-value store like Redis, whereas the leaf components might be serialized and compressed on S3.

6.3 Bridging Theory and Practice

Ultimately, this dissertation demonstrates that building successful cloud-native systems requires a deep integration of distributed systems theory with cloud engineering practice. The theoretical foundations of state persistence trade-offs, consistency models, and fault tolerance mechanisms provide the correctness guarantees and conceptual frameworks necessary for reasoning about distributed behavior. However, theory alone is insufficient; the practice of cloud engineering – optimizing for object storage latency characteristics, exploiting heterogeneous instance types, and leveraging managed services – determines whether a system is actually viable in production.

Bibliography

[1] Arrow flight. `https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/`.

[2] Databricks tpc-ds. `https://www.tpc.org/results/fdr/tpcds/databricks~tpcds~100000~databricks_sql_8.3~fdr~2021-11-02~v01.pdf`. Accessed on May 29, 2023.

[3] Mythbusting snowflake pricing! all the cool stuff you get with 1 credit. `https://medium.com/snowflake/mythbusting-snowflake-pricing-all-the-cool-stuff-you-get-with-1-credit-f3daad217a98`.

[4] An overview of end-to-end exactly-once processing in apache flink. `https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html`.

[5] Snowflake debugging info. `https://stackoverflow.com/questions/58973007/what-are-the-specifications-of-a-snowflake-server`. Accessed on May 29, 2023.

[6] Spark enhancements for elasticity and resiliency on amazon emr. `https://aws.amazon.com/blogs/big-data/spark-enhancements-for-elasticity-and-resiliency-on-amazon-emr/`.

[7] Trino fault tolerance. `https://github.com/trinodb/trino/wiki/Fault-Tolerant-Execution`.

[8] Vector data lakes, 2023. [Accessed: 29-November-2023].

[9] Index ivfpq - lancedb documentation, 2024. Accessed: 2024-08-26.

[10] What is delta lake? `https://docs.databricks.com/en/delta/index.html`, 2024.

[11] 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.

[12] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 337–350, 2015.

[13] 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.

[14] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. Nodb: efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2012.

[15] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, 2017.

[16] Amazon Web Services. Optimizing s3 performance. `https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html`, 2023.

[17] Amazon Web Services. Amazon athena, 2024. Accessed: 2024-06-29.

[18] Amazon Web Services. Amazon OpenSearch service pricing, 2024. Accessed: 2024-12-09.

[19] Amazon Web Services. Orca security's journey to a petabyte-scale data lake with Apache Iceberg and AWS analytics. AWS Big Data Blog, 2024. Accessed: December 2024.

[20] Amazon Web Services. Spot instance interruption notices. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html`, 2024. Amazon EC2 User Guide.

[21] Amazon Web Services. Amazon Simple Storage Service (S3) Documentation, Latest.

[22] Apache Parquet. Apache parquet: Columnar storage for hadoop, 2024. Accessed: 2024-07-03.

[23] Apache Software Foundation. Apache airflow, 2024.

[24] Apache Software Foundation. Apache hudi. `https://hudi.apache.org`, 2024. Accessed: 2024-08-31.

[25] Apache Software Foundation. Apache iceberg. `https://iceberg.apache.org`, 2024. Accessed: 2024-08-31.

[26] Apache Software Foundation. Apache parquet. `https://parquet.apache.org`, 2024. Accessed: 2024-08-31.

[27] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

[28] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

[29] Michael Armbrust et al. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, volume 8, 2021.

[30] 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.

[31] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. Amazon redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2205–2217, 2022.

[32] Databases at CERN. Enhancing apache spark and parquet efficiency: A deep dive into column indexes and bloom filters. *CERN Database Blog*, 2024. Accessed: 2024-08-31.

[33] Mark Atkins. Find strings within strings faster with the new Elasticsearch wildcard field, 2021.

[34] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[35] Jeff Barr. Amazon s3 update – strong read-after-write consistency. *AWS News Blog*, 2020. Accessed: 2024-08-31.

[36] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, et al. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2326–2339, 2022.

[37] Muhammad Bilal, Marco Canini, and Rodrigo Rodrigues. Finding the right cloud configuration for analytics clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 208–222, 2020.

[38] Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. Finger search in grammar-compressed strings. *arXiv preprint arXiv:1507.02853*, 2015.

[39] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Conference on Innovative Data Systems Research*, volume 5, pages 225–237, 2005.

[40] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[41] Michael Burrows. A block-sorting lossless data compression algorithm. *SRS Research Report*, 124, 1994.

[42] 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.

[43] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment*, 9(13):1401–1412, 2016.

[44] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2011.

[45] Clickhouse. Clickhouse. `https://github.com/ClickHouse/ClickHouse`, 2023. Accessed on July 10, 2023.

[46] ClickHouse, Inc. Clickhouse: Fast open-source olap dbms, 2024. Accessed: 2024-07-03.

[47] Cribl. Parquet schemas - Cribl Stream documentation, 2024. Accessed: December 2024.

[48] 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.

[49] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proceedings*

of the 2016 International Conference on Management of Data, SIGMOD '16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery.

[50] DataDog. Datadog, 2023.

[51] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[52] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[53] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)*, 17(4):1–32, 2021.

[54] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

[55] Dominik Durner, Viktor Leis, and Thomas Neumann. Exploiting cloud object storage for high-performance analytics. *Proceedings of the VLDB Endowment*, 16(11):2769–2782, 2023.

[56] Elastic. Elastic kibana, 2023.

[57] ElasticSearch NV. Elasticsearch. `https://github.com/elastic/elasticsearch`, 2024. Accessed: 2024-08-31.

[58] Facebook. Zstandard - Fast real-time compression algorithm. `https://github.com/facebook/zstd`, 2023. Original-source code available at https://github.com/facebook/zstd.

[59] Facebook, Inc. Nimble. `https://github.com/facebookincubator/nimble`, 2024. Accessed: 2024-08-28.

[60] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE, 2000.

[61] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[62] Apache Software Foundation. Apache iceberg. `https://github.com/apache/iceberg`, 2024. Accessed: 2024-05-04.

[63] Alan Gates and Daniel Dai. *Programming pig: Dataflow scripting with hadoop.* " O'Reilly Media, Inc.", 2016.

[64] Ionel Gog, Michael Isard, and Martín Abadi. Falkirk wheel: Rollback recovery for dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 373–387, 2021.

[65] Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J Puglisi. Fixed block compression boosting in fm-indexes: Theory and practice. *Algorithmica*, 81:1370–1391, 2019.

[66] Google Cloud. Use preemptible vms to run fault-tolerant workloads. `https://cloud.google.com/kubernetes-engine/docs/how-to/preemptible-vms`, 2024. Google Kubernetes Engine (GKE) Documentation.

[67] Grafana. Grafana loki oss — log aggregation system, 2023.

[68] Grafana Labs. Understanding Grafana Cloud Logs Billing. Grafana Cloud Documentation, 2024. Accessed: 2024-02-23.

[69] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. 2003.

[70] Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *2011 First International Conference on Data Compression, Communications and Processing*, pages 210–221. IEEE, 2011.

[71] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 383–394, 2005.

[72] James Holt and Leonard McMillan. Merging of multi-string bwts with applications. *Bioinformatics*, 30(24):3524–3531, 2014.

[73] Chin-Jung Hsu, Vivek Nair, Vincent W Freeh, and Tim Menzies. Arrow: Low-level augmented bayesian optimization for finding the best cloud vm. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 660–670. IEEE, 2018.

[74] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major technical advancements in apache hive. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1235–1246, 2014.

[75] InfluxData. Using parquet's bloom filters for efficient query performance. *InfluxData Blog*, 2024. Accessed: 2024-08-31.

[76] Guy Joseph Jacobson. *Succinct static data structures.* Carnegie Mellon University, 1988.

[77] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

[78] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[79] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[80] Suman Karumuri, Franco Solleza, Stan Zdonik, and Nesime Tatbul. Towards observability data management at scale. *ACM SIGMOD Record*, 49(4):18–23, 2021.

[81] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, 2018.

[82] 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.

[83] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. Btrblocks: efficient columnar compression for data lakes. *Proceedings of the ACM on Management of Data*, 1(2):1–26, 2023.

[84] Julian Labeit, Julian Shun, and Guy E Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. *Journal of Discrete Algorithms*, 43:2–17, 2017.

[85] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review*, 44(2):35–40, 2010.

[86] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173*, 2012.

[87] LanceDB. Introducing lance v2. `https://blog.lancedb.com/lance-v2/`, 2024. Accessed on [Insert access date here].

[88] Christian Lauer. Google bigquery introduces apache iceberg tables. *Medium*.

[89] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 743–754, 2014.

[90] Viktor Leis and Maximilian Kuschewski. Towards cost-optimal query processing in the cloud. *Proceedings of the VLDB Endowment*, 14(9):1606–1612, 2021.

[91] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience*, 46(11):1547–1569, 2016.

[92] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *bioinformatics*, 25(14):1754–1760, 2009.

[93] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. Streamscope: Continuous reliable distributed processing of big data streams. In *13th USENIX Symposium on Networked Systems Design and Implementation*, pages 439–453, 2016.

[94] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60:91–110, 2004.

[95] M3DB. M3: Open source metrics engine, 2023.

[96] Christos Makris. Wavelet trees: A survey. *Computer Science and Information Systems*, 9(2):585–625, 2012.

[97] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[98] James Malone. Iceberg tables: Powering open standards with snowflake innovations, 2022.

[99] 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.

[100] Alex Merced. Understanding apache iceberg delete files. *Medium*, 2022. Accessed: 2024-08-31.

[101] Mehryar Mohri. Weighted finite-state transducer algorithms. an overview. *Formal Languages and Applications*, pages 551–563, 2004.

[102] Yuta Mori. libdivsufsort: A lightweight suffix sorting library. `https://github.com/y-256/libdivsufsort`. Accessed: 2024-06-22.

[103] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging ai applications. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 561–577, 2018.

[104] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[105] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[106] Craig G Nevill-Manning and Ian H Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[107] OpenSearch. Opensearch, 2023.

[108] Ron Ortloff and Steve Herbert. Unifying iceberg tables on snowflake. `https://www.snowflake.com/blog/unifying-iceberg-tables/`, Aug 2023.

[109] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385, 1996.

[110] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. Acorn: Performant and predicate-agnostic search over vector embeddings and structured data. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024.

[111] Guilherme Penedo, Hynek Kydlíček, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, Thomas Wolf, et al. The fineweb datasets: Decanting the web for the finest text data at scale. *arXiv preprint arXiv:2406.17557*, 2024.

[112] Rahul Potharaju, Terry Kim, Eunjin Song, Wentao Wu, Lev Novik, Apoorve Dave, Andrew Fogarty, Pouria Pirzadeh, Vidip Acharya, Gurleen Dhody, et al. Hyperspace: The indexing subsystem of azure synapse. *Proceedings of the VLDB Endowment*, 14(12):3043–3055, 2021.

[113] Rahul Potharaju, Terry Kim, Wentao Wu, Vidip Acharya, Steve Suh, Andrew Fogarty, Apoorve Dave, Sinduja Ramanujam, Tomas Talius, Lev Novik, et al. Helios: hyperscale indexing for the cloud & edge. *Proceedings of the VLDB Endowment*, 13(12):3231–3244, 2020.

[114] Taivo Pungas. Gpt-3.5 and gpt-4 api response time measurements - fyi, May 2023.

[115] Qdrant, Inc. Qdrant. `https://github.com/qdrant/qdrant`, 2024. Accessed: 2024-08-31.

[116] Quickwit. Quickwit, 2023.

[117] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019.

[118] Kirk Rodrigues, Yu Luo, and Ding Yuan. Clp: Efficient and scalable search on compressed text logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 183–198, 2021.

[119] Delta rs Contributors. delta-rs: Native rust bindings for delta lake. `https://github.com/delta-io/delta-rs`, 2024. Accessed: 2024-05-27.

[120] Mingwei Samuel, Joseph M Hellerstein, and Alvin Cheung. *Hydroflow: A Model and Runtime for Distributed Systems Programming.* PhD thesis, Master's thesis. EECS Department, University of California, Berkeley. http . . . , 2021.

[121] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. Predicate caching: Query-driven secondary indexing for cloud data warehouses. 2024.

[122] SentinelOne. DataSet, 2023. Accessed: 2023-11-27.

[123] Amazon Web Services. Security data management - amazon security lake, 2021.

[124] 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*, pages 1802–1813. IEEE, 2019.

[125] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. Push versus pull-based loop fusion in query engines. *Journal of Functional Programming*, 28:e10, 2018.

[126] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.

[127] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, et al. Dbos: a dbms-oriented operating system. *Proceedings of the VLDB Endowment*, 15(1):21–30, 2021.

[128] Splunk. How to extract bunch of UUIDs from a string using regex, 2022.

[129] Splunk. Splunk, 2023.

[130] Splunk Inc. Indexing and search architecture, 2024. Accessed: 2024-07-03.

[131] Sagar Sumit. Asynchronous indexing using hudi.

[132] Sumo Logic. What you should know about Datadog Flex Logs and Pricing. Sumo Logic Blog, 2023. Accessed: 2024-02-23.

[133] Delta Lake Team. Introducing deletion vectors in delta lake: Streamlined data deletion for faster queries. *Delta.io Blog*, 2023. Accessed: 2024-08-31.

[134] Trino Software Foundation. Trino. `https://trino.io`, 2024. Accessed: 2024-08-31.

[135] Alexandru Uta, Bogdan Ghit, Ankur Dave, Jan Rellermeyer, and Peter Boncz. In-memory indexed caching for distributed data processing. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 104–114. IEEE, 2022.

[136] Joris van der Walker. The burrows-wheeler transform. `https://curiouscoding.nl/posts/bwt/`, 2023. Accessed on 2024-10-20.

[137] Vantage. Aws ec2 r6i.xlarge on-demand instance pricing. `https://instances.vantage.sh/aws/ec2/r6i.xlarge`, 2024. Accessed on [Insert Access Date].

[138] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation*, pages 449–462, 2020.

[139] Dalin Wang, Feng Zhang, Weitao Wan, Hourun Li, and Xiaoyong Du. Finequery: Fine-grained query processing on cpu-gpu integrated architectures. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 355–365. IEEE, 2021.

[140] Mengzhao Wang, Lingwei Lv, Xiaoliang Xu, Yuxiang Wang, Qiang Yue, and Jiongkang Ni. An efficient and robust framework for approximate nearest neighbor search with attribute constraint. *Advances in Neural Information Processing Systems*, 36, 2024.

[141] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for {Fine-Grained} tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation*, pages 671–686, 2021.

[142] Ziheng Wang. Quokka documentation. `https://marsupialtail.github.io/quokka/`.

[143] Ziheng Wang, Emanuel Adamiak, and Alex Aiken. A model for query execution over heterogeneous instances. In *CIDR*, 2024.

[144] Ziheng Wang and Alex Aiken. Efficient fault tolerance for pipelined query engines via write-ahead lineage. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 436–448. IEEE, 2024.

[145] Ziheng Wang, Sasha Krassovsky, Conor Kennedy, Alex Aiken, Weston Pace, Rain Jiang, Huayi Zhang, Chenyu Jiang, and Wei Xu. Rottnest: Indexing Data Lakes for Search . In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 1814–1827, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[146] Ziheng Wang, Junyu Wei, Alex Aiken, Guangyan Zhang, Jacob O. Tørring, Rain Jiang, Chenyu Jiang, and Wei Xu. Logcloud: Fast search of compressed logs on object storage. *Proceedings of the VLDB Endowment*, 18(8):2362–2370, 2025.

[147] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. Loggrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 452–468, 2023.

[148] Junyu Wei, Guangyan Zhang, Yang Wang, Zhiwei Liu, Zhanyang Zhu, Junchao Chen, Tingtao Sun, and Qi Zhou. On the feasibility of parser-based log compression in {Large-Scale} cloud systems. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 249–262, 2021.

[149] Shiyan Xu and Sivabalan Narayanan. Record level index: Hudi's blazing fast indexing for large-scale datasets, 2023. Accessed: 2024-04-08.

[150] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 452–465, 2017.

[151] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28, 2012.

[152] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[153] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. An empirical evaluation of columnar storage formats. *arXiv preprint arXiv:2304.05028*, 2023.

[154] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *Proceedings of the VLDB Endowment*, 11(11):1522–1535, 2018.

[155] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. Tadoc: Text analytics directly on compression. *The VLDB Journal*, 30:163–188, 2021.

[156] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2653–2666, 2021.

[157] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R Lyu. Loghub: A large collection of system log datasets for ai-driven log analytics. *arXiv e-prints*, pages arXiv–2008, 2020.