TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Bachelor's Thesis in Informatics

Enhancing Distributed Low-Cost Object Storage with Near Storage Computation

Linus Erich Weigand

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

Bachelor's Thesis in Informatics

Enhancing Distributed Low-Cost Object Storage with Near Storage Computation

Verbesserung eines verteilten kostengünstigen Objektspeichers mit Speicher nahen Berechnungen

Author:	Linus Erich Weigand
Supervisor:	Prof. Dr. Viktor Leis
Advisor:	Maximilian Kuschewski, M.Sc.
Submission Date:	05.03.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Bachelor's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Munich, 05.03.2025 Linus Erich Weigand

Acknowledgments

I would like to express my deepest gratitude to my supervisor, M.Sc. Maximilian Kuschewski, whose insightful guidance, inspiring meetings, and generous sharing of ideas, concepts and especially time have played a key role in shaping this work. I am immensely grateful for the motivating discussions and continuous support that helped me refine my understanding of the research topic. I would also like to extend my sincere thanks to Prof. Dr. Viktor Leis for granting me the opportunity to pursue my bachelor thesis at his department. Finally, I would like to thank the Chair of Decentralized Information Systems and Data Management for their financial support, covering all associated costs, including the AWS EC2 expenses. Their assistance and resources were invaluable in carrying out the practical components of this thesis.

Abstract

The shift toward cloud-based analytics has led to the adoption of data lakes on blob storage services such as Amazon S3, facilitating seamless data sharing and system interoperability. To accommodate this paradigm, many data warehouses have embraced a disaggregated storage and compute architecture. However, this model is suboptimal for the predominantly low-selectivity queries encountered today. Specifically, the necessity for storage nodes to transmit complete datasets, even when compute nodes subsequently filter out large portions of the data, results in significant disk, network, and CPU overhead, thereby increasing both query latency and operational cost. In this work, we present a distributed architecture and propose two approaches that enable near-storage computation by pre-filtering data at the storage node. Our methods leverage the metadata inherent in the Parquet file format to prune row groups during read operations, effectively reducing I/O demands, lowering compute costs, and enhancing throughput.

Contents

1 Introduction

Data Warehousing is Moving to the Cloud. Organizations continuously collect and analyze growing datasets, with a significant portion now being stored in public cloud platforms such as Amazon AWS, Microsoft Azure, and Google Cloud. To process these datasets, many customers leverage cloud-native data warehousing systems, including Snowflake [1], Databricks [2], Amazon Redshift [3], Microsoft Azure Synapse Analytics [4], and Google BigQuery [5, 6]. A key shift in cloud data warehousing is the disaggregation of storage and compute, where data resides in distributed cloud object stores such as S3, while compute resources can be elastically provisioned as needed. This architecture, initially introduced by BigQuery and Snowflake, has influenced even systems like Redshift, which originally followed a horizontally partitioned, shared-nothing architecture but is now transitioning toward disaggregated storage [7, 8].

Inefficient Data Loading and Filtering. Traditional cloud-native data warehousing systems typically load complete datasets from storage nodes into compute nodes before any filtering occurs [9]. For low-selectivity queries, where only a small subset of the data is required, this design is highly inefficient: large volumes of data are transferred and processed only to have most of it discarded. This approach not only incurs unnecessary disk I/O but also wastes network bandwidth on data transfers that could be avoided. The resulting inefficiencies lead to increased query latency and higher operational costs, highlighting the need for architectures that enable pre-filtering directly at the storage node level.

Pre-Filtering at the Storage Nodes. A better approach involves performing best-effort pre-filtering directly at the storage nodes during data extraction. In this paper, we introduce two solutions that selectively filter row groups and columns from Parquet files prior to their transmission to the compute nodes. These methods are specifically designed to minimize overall workload cost by reducing disk I/O, network bandwidth, and computational overhead. By leveraging the intrinsic metadata provided by the Parquet format, our approach enables the selective exclusion of unnecessary data before initiating read operations. Additionally, we present a metadata-storage node architecture accompanied by a cost analysis that compares an EC2 instance-based distributed file system with conventional storage systems, such as AWS S3.

Contributions. This work consists of the following contributions: (1) A cost viability analysis comparing Amazon S3 Standard with an EC2-based solution, discussed in chapter 3. (2) An in-depth microbenchmark study that characterizes network, disk, and CPU performance on low storage cost d3en EC2 instances, also described in section 3. (3) A detailed analysis of real-world datasets, including the Snowset dataset from Snowflake and the Thesios dataset from Google Storage Systems, to derive realistic query selectivity distributions and retrieval rates, evaluated in section 4. (4) A metadata-driven pruning technique that leverages Parquet file metadata for row group and column filtering, as well as row filtering and aggregation, proposed in chapter 6. (5) An advanced direct byte-range access approach that minimizes CPU and memory usage by storing minimal metadata in memory and reading only the necessary data from disk, presented in chapter 7. (6) A comprehensive evaluation in chapter 8, spanning throughput, CPU, and memory metrics for both approaches under varied selectivity, row group conditions and techniques. In the following chapter 2 we will introduce background information and discuss related work. The approaches discussed are open source and available at https://github.com/LinusWeigand/parquet-near-storage-compute.

2 Background and Related Work

Motivation. This chapter is structured to systematically motivate the key components that underpin our proposed solution. We begin by discussing EC2 instances, which form the cost-effective and scalable foundation of our distributed object storage system. Next, we examine object stores to highlight their central role in managing large datasets in cloud environments. The Parquet file format is then introduced, emphasizing its metadata features that enable near-storage pre-filtering, a critical capability for our approach. We further explore the conventional S3 solution to underscore its limitations in cost and performance, thereby justifying the need for our alternative EC2-based architecture. Finally, we review related work to position our contributions within the broader research context.

EC2 Instances. Within the AWS cloud ecosystem, Amazon EC2 instances offer scalable, on-demand virtualized compute resources. In our work, we leverage cost-effective d3en EC2 instances as the foundation for our distributed object storage system. These instances provide not only cost-effective storage but also spare CPU and memory resources that enable near-storage pre-filtering of Parquet files, thereby enhancing query throughput and reducing unnecessary data transfers.

Object Stores. In modern cloud architectures, object stores serve as scalable repositories for unstructured data by managing content as discrete objects rather than traditional files or blocks. Their design emphasizes durability, flexibility, and easy metadata access, which has made them a cornerstone for services like S3. In our paper, we leverage these principles while addressing inherent inefficiencies, such as unnecessary data transfers, by proposing an alternative distributed object storage system built on EC2 instances with near-storage pre-filtering capabilities.

Parquet File Format. The Parquet file format is a columnar storage format specifically designed to enhance the efficiency of data processing and analytics. Data within a Parquet file is organized into row groups, which internally consist of multiple column chunks. Each row group contains all columns, ensuring that the number of rows remains consistent across all column chunks. To facilitate selective data access and predicate pushdown filtering, each column chunk is accompanied by metadata stored in the file footer. This metadata includes min-max values, among other statistics, allowing queries to efficiently eliminate irrelevant data at read time. Figure 2.1 illustrates this structural design, depicting the file format progression from left to right. On the

left, row groups and their corresponding column chunks are shown, while on the right, the file footer maintains references to each row group and column chunk. These references contain file offsets and lengths, enabling precise data retrieval and optimizing read performance by avoiding unnecessary I/O operations.

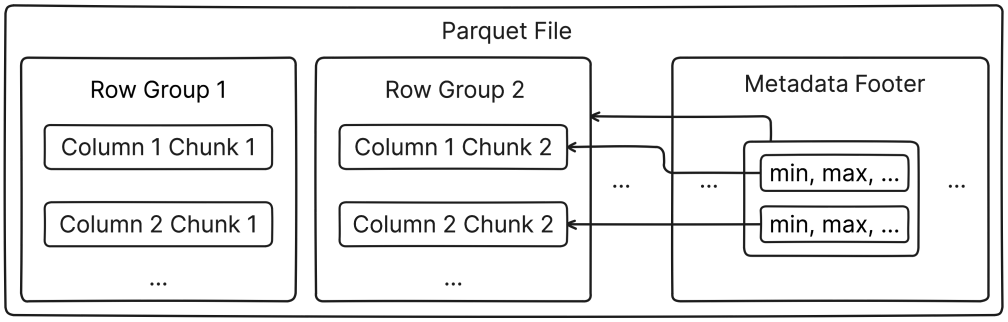

Figure 2.1: Parquet File Format

S3 on Hard Disk Drives. Object stores like Amazon S3 offer significant abstraction and scalability, while fundamentally relying on arrays of commodity hard disk drives for data persistence. Most warehouse solutions use these as a storage layer, extracting data to compute nodes when filtering is needed. This architecture often necessitates reading large portions or entire objects, even when only a small fraction of the data is relevant for a query. Consequently, this approach raises an important question: How much of the data read from object storage is actually utilized in downstream computations? Addressing this inefficiency through near-storage compute strategies, such as pre-filtering data at the storage nodes, has the potential to substantially reduce unnecessary disk I/O, network traffic, and overall system overhead. In the following chapter 3 we will discuss the viability and feasability of a distributed file system based on d3en EC2 instances, by first comparing storage costs to S3 and second examining the throughput limits of d3en instances using microbenchmarks. In chapter 4 we will then analyse real worl datasets to estimate throughput requirements of in-production storage systems.

2.1 Related Work

Near Storage Computation and Distributed File Systems There is a large body of work on near storage computation in warehouses and distributed file systems. Below we discuss a selection that relates most closely to our approach.

S3 Select. Enabling server-side query execution within Amazon S3, *S3 Select* allows users to filter, project, and aggregate data directly at the storage layer, reducing data transfer to compute nodes. It introduces lightweight query processing within S3 storage nodes, with only a limited subset of SQL [10]. It supports column projection, filtering and aggregation as well as smaller features such as numeric & string functions, logical operaters and type castings. It operates on *CSV* files and streams the data from disk. While reading row by row, it discards filtered columns and rows, and only returns cells which match the specified query. While it does not lead to reduced I/O or compute cost, it does save of network bandwith.

Parquet Select. With the addition of Parquet files to S3 Select, the extension *Parquet Select* enables efficient, in-place querying of Parquet files stored in AWS S3 [11]. Similar to our approach it leverages column pruning as well as predicate pushdown to skip row groups based on the parquet metadata's min, max values. Queries are streamed from disk, without loading the full file into memory. Row group filtering and column projection occur as data is read. Despite operating on parquet files, results are always returned in *CSV* format. The reduction in I/O leads to a decrease in network and disk overhead as well as compute cost.

Conflicting Pricing Model. We think that the limited adoption of *S3 Select* and its extension *Parquet Select* is influenced by AWS's pricing structure. Under this model, costs are mainly driven by the volume of data scanned and the number of requests, while the cost associated with the data actually returned is comparatively minimal [12]. This pricing scheme may not offer a significant financial incentive, particularly when complex queries still necessitate the use of external compute resources such as *EC2*, *Lambda*, or *Athena*. Moreover, querying data through services like Athena or Redshift Spectrum, which provide broader SQL support and often keep results within the same availability zone to avoid additional egress costs, might be in many cases more cost-effective. Consequently, *S3 Select* finds its cost benefits confined to niche scenarios involving simple queries that are directly returned to the client, saving on egress costs.

Redshift Aqua. Redshifts native columnar storage format, enables preemptively discarding unnecessary data before reaching the compute node by using column statisitics and zone maps, inherent to its metadata [13]. Column statistics provide insights such histograms, value distribution and frequency count. This helps estimating query selectivity allowing AQUA to more aggressively skip blocks or parts of blocks. Additionally each block in the columnar storage is accompanied by a zone map, which stores the minimum and maximum values of each column in that block, similar to parquet. When a query with a filtering predicate is issued, AQUA inspects these zone maps to determine if the block could possibly contain any rows that satisfy the predicate. If the predicates condition lies entirely outside the range stored in a blocks zone map, AQUA can safely skip reading that block [14].

Snowflake. A pioneering advancement in data warehousing architecture, the disaggregation of compute and storage facilitates independent scaling of resources, enhancing elasticity and operational efficiency. In Snowflake's model, the storage layer is maintained as immutable files within Amazon S3, ensuring high durability and availability. Query execution in this architecture follows a structured sequence designed to maximize efficiency through pre-read data pruning. First, queries are submitted to the Cloud Services layer, where they are parsed, optimized, and scheduled for execution. Next, the designated compute nodes interact with the metadata service, which provides critical information, including file locations in Amazon S3, min-max values for pruning, and column-wise offset and length metadata, referred to as zone maps, to facilitate selective access. With this metadata, the compute nodes determine the precise byte ranges required for processing, generating offset-length tuples that are then requested from Amazon S3. The storage layer, devoid of any compute functionality, responds by delivering only the requested byte ranges, thereby minimizing unnecessary data transfer. The system leverages a hybrid PAX format, which groups columnar data within each file, enabling efficient predicate-based filtering and projection pruning before data retrieval, further optimizing query performance [15, 16].

3 Viability and Feasability Analysis

3.1 Cost Analysis

We first examine the cost viability of our approach. We will compare costs solely against the Amazon S3 Standard Tier. Other tiers (e.g., S3 Glacier, S3 Glacier Deep Archive) impose additional constraints, such as minimum billing durations, extra retrieval costs, or per-request fees, which complicate direct comparisons. The S3 Standard Tier employs a piecewise pricing model that decreases with increasing stored volume.

Overall Estimated Storage Requirements. We will calculate the cost on the basis of storage need. In later paragraphs we will look at the Snowset dataset more closely and how we estimate warehouse sizes. For now we estimated Snowflakes storage needs to be 172.5 PB at the time of dataset creation which is March 7th 2018. We therefore will estimate the monthly costs of storing this 172.5 PB using (1) Amazon S3 Standard and (2) and EC2-based solution.

Cost Estimation for S3 Standard. We apply S3's pricing model and compute the total:

$$\begin{aligned} C_{S3} &= (50{,}000\text{ GB}) \times 0.023 \;+\; (450{,}000\text{ GB}) \times 0.022 \;+\; (172{,}000{,}000\text{ GB}) \times 0.021 \\ &= \$3{,}623{,}050 \text{ per month.} \end{aligned}$$

Cost Estimation for an EC2-Based Approach.

We evaluate an alternative approach using EC2 `d3en.4xlarge` instances. Among available instance types, the `d3en` family exhibits the lowest cost per gigabyte-month,[1] and the `4xlarge` variant provides consistent bandwidth (where smaller variants only guarantee up to a certain network throughput). Each `d3en.4xlarge` instance provides eight ephemeral hard disks, totaling: 111,840 GB per instance. To store the required 172,500,000 GB, the total number of instances N is:

$$N = \left\lceil \frac{172{,}500{,}000\,\text{GB}}{111{,}840\,\text{GB/instance}} \right\rceil \approx 1543 \text{ instances.}$$

Monthly Instance Cost. For a reserved Linux `d3en.4xlarge` instance, the hourly rate in Frankfurt is approximately $1.7460. The monthly cost of a single instance is:

$$C_{\text{instance}} = 1.7460 \times 24 \times \frac{365}{12} \approx \$1274.58.$$

[1] Based on AWS on-demand or reserved instance pricing at the time of writing.

Total Monthly Cost for Plain Storage. Multiplying by N instances, the total monthly cost for plain (unencoded) storage is:

$$C_{\text{EC2-plain}} = N \times C_{\text{instance}} = 1543 \times 1274.58 \approx \$1{,}966{,}676.94$$

which is approximately 54% of the S3 cost. However, it should be noted that this configuration does not provide the same durability guarantees as S3 Standard.

Cost Impact of Erasure Coding Overhead. S3 internally applies a Reed–Solomon $(10, 4)$ erasure-coding scheme, implying a storage overhead of 40%. In S3 Standard, the user cost model is abstracted from this overhead. In an EC2-based solution, such overhead must be explicitly accounted for by provisioning additional storage. Hence, the total storage requirement becomes:

$$175{,}500{,}000 \text{ GB} \times 1.4 = 245{,}700{,}000 \text{ GB}$$

Recomputing the instance count:

$$N_{\text{EC2-EC}} = \left\lceil \frac{245{,}700{,}000 \text{ GB}}{111{,}840 \text{ GB/instance}} \right\rceil \approx 2197 \text{ instances.}$$

The corresponding monthly cost is:

$$C_{\text{EC2-EC}} = N_{\text{EC2-EC}} \times C_{\text{instance}} = 2197 \times 1274.58 \approx \$2{,}800{,}252.26,$$

which is about 77% of the S3 cost.

Result Discussion. Our cost estimation analysis indicates that operating a distributed object store entirely on d3en EC2 instances is a viable alternative to Amazon S3 in terms of storage costs. The findings suggest that, under the given assumptions, this approach can achieve significant cost savings while maintaining scalability. In the following sections, we shift our focus to feasibility, examining whether the available resources allow for the efficient operation of a distributed object store and whether the estimated throughput meets the requirements of real-world workloads.

3.2 Microbenchmarks

The purpose of this microbenchmark study is to evaluate the fundamental resource and bandwidth constraints of our decentralized object storage system to get a first feasability estimate. By complementing theoretical performance figures with empirical measurements, we derive practical upper limits on system throughput and resource utilization. The insights gained will enable accurate prediction of near-storage computational capacity. We structure our benchmarks into four key measurements: (1) Network bandwidth (send and receive operations). (2) Disk throughput (read and write operations). (3) Combined disk and network performance to evaluate synchronozation costs. (4) Overhead analysis of TCP versus HTTP communication.

Test setup. All experiments are conducted on an AWS EC2 `d3en.4xlarge` instance, chosen due to its cost-effectiveness for high-storage applications. The instance is equipped with: (1) A 2nd Generation Intel Xeon Scalable (Cascade Lake) CPU with 16 cores clocked at 3.1 GHz. (2) 64 GiB of RAM. (3) Eight hard disks with a combined capacity of 111,840 GB, each rated at 250 MiB/s theoretical throughput. The benchmarking programs are implemented in Rust and compiled using Cargo 1.82.0 on Amazon Linux 2. The `tokio` library is used for parallelization and asynchronous I/O, while network operations leverage `hyper` (HTTP) and `reqwest` (HTTP client). Before each disk benchmark, the page cache is cleared using:

```
sync && echo 1 | sudo tee /proc/sys/vm/drop_caches
```

Disk Setup. A RAID 0 array is created using `mdadm` to combine the eight hard disks into a single logical volume. The array is formatted with XFS to maximize throughput and concurrency.

Measurement Methodology. Each benchmark measures the following system metrics: (1) CPU utilization: aggregated from `/proc/stat`. (2) Memory consumption: recorded using `free -m`. (3) Disk throughput: measured via `/proc/diskstats`. (4) Network throughput: recorded from `rx_bytes` and `tx_bytes` in `/sys/class/net/ens5/statistics/`.

3.2.1 Network Throughput Measurement

To evaluate network performance, we implemented a high-performance client-server model using TCP. The setup consists of two `d3en.4xlarge` instances within the same virtual private cloud (VPC) to ensure minimal latency. The client establishes eight parallel TCP connections and transmits 256 MiB blocks of random data to the server, which discards the received bytes immediately.

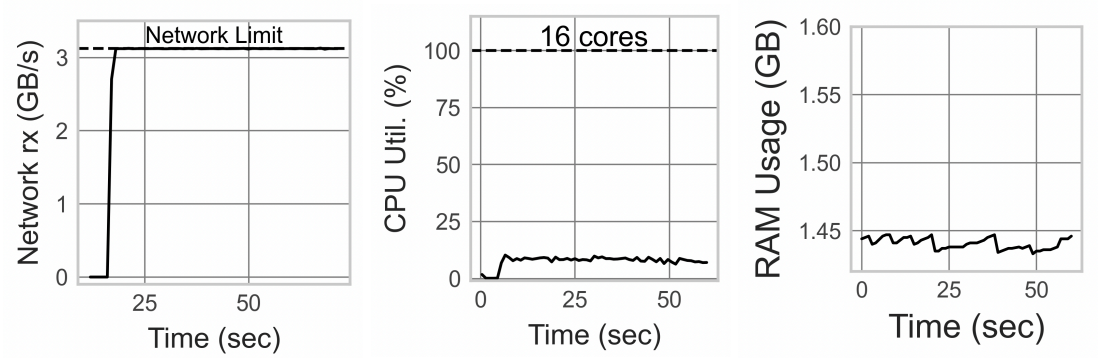

Figure 3.1: Server resource utilization while receiving bytes over TCP

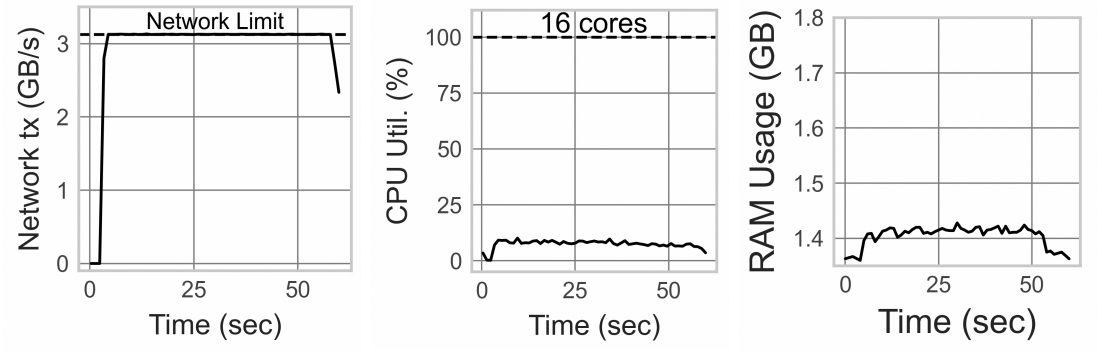

Figure 3.2: Client resource utilization while sending bytes over TCP

Result Discussion. Empirical measurements show that both client and server fully utilize the theoretical network bandwidth of 3.125 GB/s. CPU utilization during transmission averages 7.98% on the server and 11.75% on the client. Memory usage remains negligible, with a peak increase of 40 MiB on the client side. This indicates that network I/O is neither CPU nor Memory intensive, and approximately 88% of CPU resources an almost all Memory remain available for near-storage computation.

3.2.2 Disk Throughput Measurement

Two separate disk benchmarks are executed: (1) Sequentially reading a large file in a continuous loop. (2) Sequentially writing random data to a file. A block size of 512 KiB is determined to be optimal. Due to the sequential nature of HDDs, a single-threaded approach achieves the best performance.

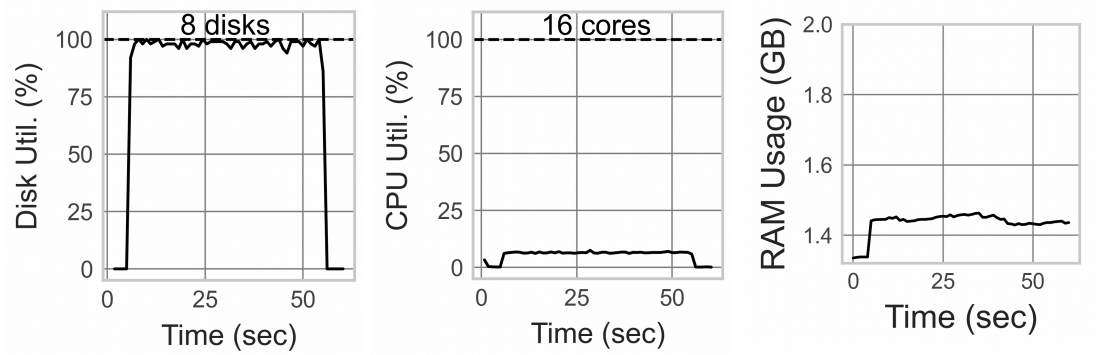

Figure 3.3: Resource utilization while reading bytes from disk

Result Discussion. Empirical measurements reveal a near-maximal utilization of disk throughput. The system achieves a read throughput of 1.923 GiB/s and a write throughput of 1.978 GiB/s. CPU utilization is measured at 6.52% for read operations and

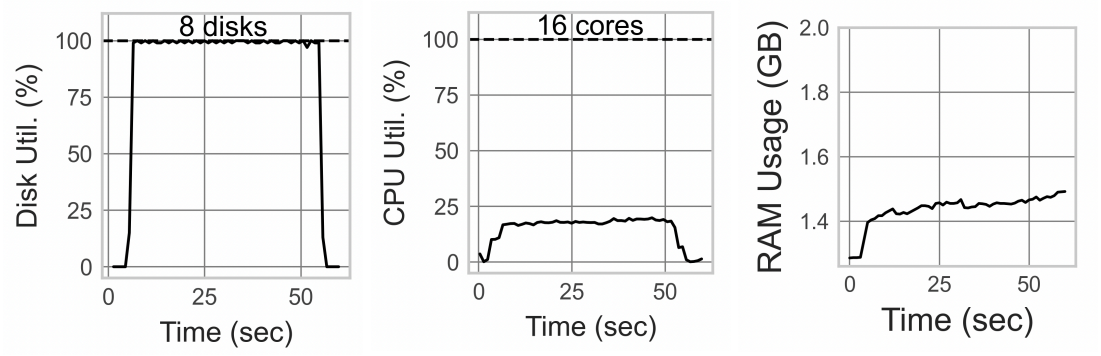

Figure 3.4: Resource utilization while writing bytes to disk

31.82% for write operations. Memory consumption spikes for both read and write by roughly 102MB and 125MB, respectively, with disk utilization reaching 98.17% for reads and 99.48% for writes.

3.2.3 Combined Network and Disk Benchmark

To simulate real-world workloads, a program is designed to receive data over eight parallel TCP streams and write it to disk. Each TCP stream writes to a dedicated file, with a producer-consumer architecture managing data transfer between network and disk subsystems.

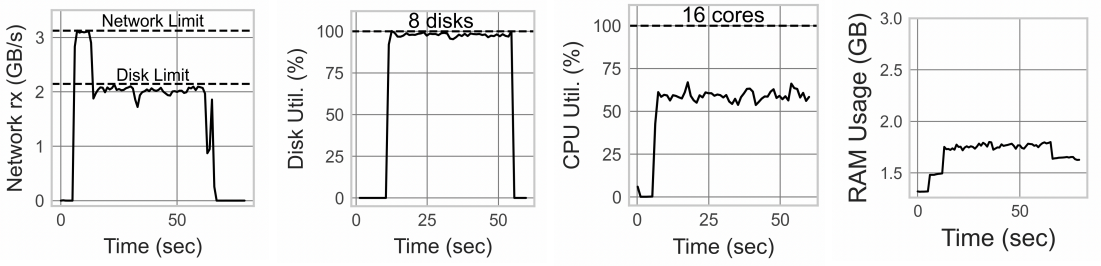

Figure 3.5: Resource utilization receiving via TCP and writing bytes to disk

Result Discussion. The observed disk utilization of 99.59% and the sharp decrease of network throughput once the producer-consumer queues are full, confirms that the system is disk-bound. CPU utilization averages 63.42%, significantly higher than the sum of independent network and disk benchmarks (43.23%). Memory increases by 401 MB. This increase is attributed to the overhead of managing multiple producer-consumer queues. The write throughput is measured at 1.901 GiB/s, slightly lower than the standalone disk benchmark due to synchronization overhead.

3.2.4 HTTP Overhead Evaluation

To quantify the additional cost of HTTP over TCP, the server from section 3.2.1 is reimplemented using HTTP instead of TCP.

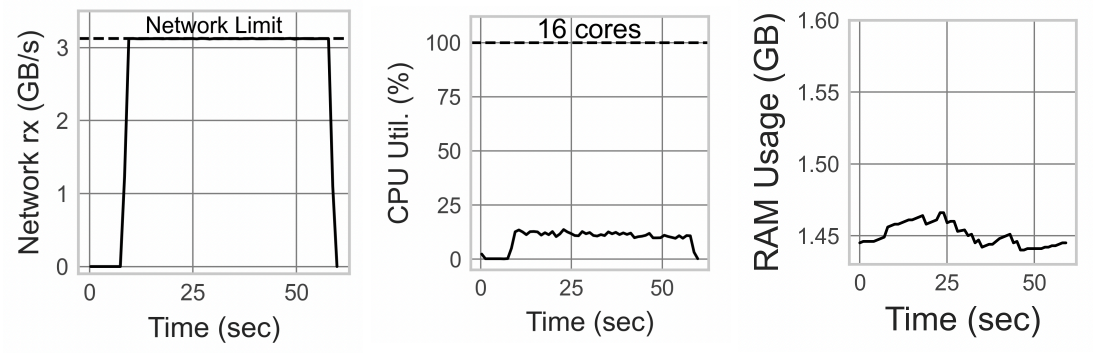

Figure 3.6: Server resource utilization while receiving bytes via HTTP

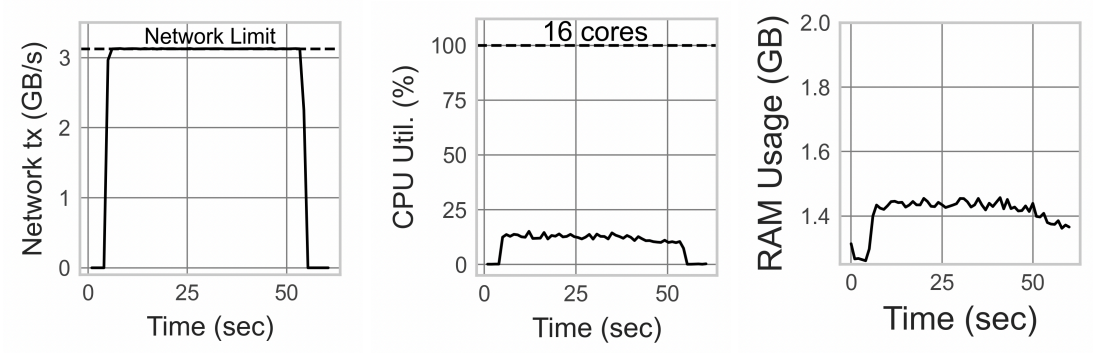

Figure 3.7: Client resource utilization while sending bytes via HTTP

Result Discussion. While network throughput remains stable at 3.125 GiB/s, CPU utilization increases by 43% on the client and 4.8% on the server. Memory usage remains consistent, indicating that HTTP overhead primarily affects CPU performance. This suggests that mostly computationally idle servers are justified to use HTTP, but applications needing the CPU for near-storage-computation should prioritize TCP where feasible to reduce computational overhead.

Resource Constraint Expectations. We provide a preliminary quantitative analysis of our final approaches resource constraints in terms of maximum throughput and residual CPU and memory capacity. Our measurements indicate that network overhead introduces an additional CPU utilization of approximately 7.98% with negligible impact on memory. For disk read operations, we achieve a maximum throughput of

1.923 GiB/s at a CPU overhead of 6.52%, whereas write operations incur a CPU overhead of 31.82% and an extra 102 MB of memory usage. When combining disk I/O and network synchronization overhead, we estimate an overall CPU utilization of 63.42%, a disk throughput of 1.901 GiB/s, and a memory increase of roughly 401 MB, keeping total memory usage below 2 GB. Consequently, for read queries, the effective CPU load is approximated by $63.42\% - 31.82\% + 6.52\% \approx 38.12\%$ Assuming a system memory capacity of 64 GB, this leaves approximately 62 GB available for near-storage computation, and the spare CPU capacity is roughly $100\% - 38.12\% \approx 61.88\%$.

3.3 Conclusion

From a pure storage-cost perspective, the EC2-based solution remains competitive even when accounting for a 40% erasure-coding overhead. However, while cost determines viability, feasibility depends on factors such as query throughput and retrieval efficiency. While data durability is not the focus of this work, we concentrate on optimizing query throughput by evaluating system constraints and potential performance improvements. Through our microbenchmark analysis, we have established key performance characteristics: (1) Network bandwidth can be fully utilized with minimal CPU and memory overhead. (2) Disk throughput approaches its theoretical limits, though write operations incur significantly higher CPU overhead. (3) Combined disk and network workloads are predominantly disk-bound, with additional CPU overhead resulting from synchronization. (4) HTTP introduces measurable CPU overhead compared to TCP, although its impact on memory remains negligible. In addition, our resource constraint analysis indicates that for read queries the effective CPU utilization is approximately 38.12%, leaving around 61.88% of the CPU resources available for near-storage-compute tasks, with nearly 62 GB of memory remaining accessible. These empirical performance limits highlight the importance of I/O reduction as the primary optimization target to enhance overall query throughput. In subsequent sections, we examine real-world query selectivity distributions to identify further optimizations, which will be integrated into our final evaluation in chapter 8.

4 Real-World Datasets

Requirements. Query requests to a decentralized object storage system vary heavily on their request pattern. To evaluate our approaches and reach our conclusions on throughput performance, we first need to examine real-world queries on two metrics: (1) How many times a stored GB will be expected to be retrieved on average per month and (2) What the selectivity distribution on real storage looks like. We can then come to a prelimitary conclusion if the measured throughput in chapter 3.2 of a EC2 instance based storage system suffices to meet real world throughput requirements. For later presented optimizations we can calculate an expected throughput, based on measured throughput data points corresponding to different selectivity levels and the probability distribution of query selectivities from real storage systems. This will inform us if the current approach meets the necessary throughput requirements. We will examine two datasets in total: (1) The Snowset dataset from Snowflake and (2) The Thesios dataset from Google Storage Systems.

4.1 The Snowset Dataset

The large real-world collection of queries we chose to focus on first is the `Snowset Dataset` [17, 18]. This dataset is an aggregation of queries submitted to the Snowflake data warehouses over a time period of 14 days from Feb 21st 2018 to March 7th 2018. It aggregates Snowflakes internal server metrics into individual queries. With 93 attributes per query it contains a plethora of information such as `total end-to-end query time`, `total bytes scanned` and a wide range of other system metrics such as time spend in different regions of the Snowflake's architecture. It contains approximately 70 Million queries from Snowflake's customers. We thus expect its contents to be representative of what we can expect the throughput requirements to be in a real-world storage system.

Estimating Warehouse Sizes. We aim to estimate the size of each warehouse based on query execution metrics. The key factors in this estimation are:

(1) `Query Size` (q_i): The total amount of data read by each query. (2) `Warehouse Size` (W_i): The estimated total data size of the warehouse from which the query reads.

Table 4.1: Relevant Snowset Query Attributes

Column	Description
warehouseId	Identifier of warehouse in which the query ran
scanBytes	Total number of bytes scanned from files
scanAssignedFiles	Total number of files to be scanned after pruning
scanOriginalFiles	Total number of files in the table before pruning

Deriving Query Size. The size of data scanned by a query can be directly obtained from the `scanBytes` column in the Snowset dataset:

$$q_i = \text{scanBytes}_i \tag{4.1}$$

where: (1) q_i represents the amount of data read by query i. (2) scanBytes_i is the total number of bytes scanned for query i.

Estimating Warehouse Size. For each query, we estimate the warehouse size using the following formula:

$$W_i = \frac{\text{scanBytes}_i}{\text{scanAssignedFiles}_i} \times \text{scanOriginalFiles}_i \tag{4.2}$$

where: (1) W_i is the estimated size of the warehouse relevant to query i.
(2) $\text{scanAssignedFiles}_i$ represents the number of files scanned by query i in the warehouse. (3) $\text{scanOriginalFiles}_i$ represents the total number of files currently present in the warehouse. This formula assumes that the scanned data is proportionally distributed across assigned files, allowing us to extrapolate the total warehouse size.

Aggregating Warehouse Size Estimates. To derive a final estimate for each warehouse, we aggregate the warehouse sizes across all queries for a given `warehouseId` and take the `maximum estimated size`:

$$\hat{W} = \max_i W_i \tag{4.3}$$

where: (1) $\hat{W}$ is the final estimated warehouse size for a given warehouse. (2) The `maximum` function ensures we account for the largest observed warehouse size, preventing underestimation due to partial scans.

SQL Query for Warehouse Size Estimation. The following SQL query implements this estimation by computing W_i for each query and taking the `maximum warehouse estimate per warehouseId`:

```sql
SELECT
  warehouseId,
  (
    NULLIF(
      MAX(
        (scanBytes / NULLIF(scanAssignedFiles, 0)) *
            scanOriginalFiles
      ),
      0
    )
  ) AS estimated_warehouse_size
FROM
  'snowset-main.parquet/*.parquet'
GROUP BY
  warehouseId;
```

This approach provides a robust estimate of warehouse sizes based on observed query execution data.

Approximating Throughput Requirements. We aim to estimate how frequently stored data is retrieved from warehouses over time, measured as the number of times a stored gigabyte (GB) is retrieved per month. Let: (1) q_i be the total number of bytes scanned by query i. (2) W_j be the estimated size of warehouse j. (3) w be the total number of warehouses. For each warehouse j, we compute the `data retrieval rate` as the ratio of total bytes scanned across all queries that ran on warehouse j, divided by the estimated warehouse size:

$$R_j = \frac{\sum_{i \in Q_j} q_i}{\hat{W}_j} \tag{4.4}$$

where: (1) R_j represents the `retrieval rate` for warehouse j. (2) Q_j is the set of all queries that have accessed warehouse j. (3) $\sum_{i \in Q_j} q_i$ represents the total scanned bytes across all queries in warehouse j.

Weighted Average Retrieval Rate. To obtain a global estimate over all warehouses, we compute the `weighted average retrieval rate` across all warehouses, where each warehouse contributes proportionally to its size:

$$R_{\text{avg}} = \frac{\sum_{j=1}^{w} R_j \cdot \hat{W}_j}{\sum_{j=1}^{w} \hat{W}_j} \tag{4.5}$$

where: (1) R_{avg} is the overall weighted average retrieval rate. (2) The numerator sums the total retrieval activity weighted by warehouse sizes. (3) The denominator normalizes the result by the total storage across all warehouses. Since we are interested in the

retrieval rate per month, we convert the computation from a 14-day period to a full year and normalize it to a monthly scale:

$$R_{\text{monthly}} = \frac{R_{\text{avg}}}{14} \times 365 \times \frac{1}{12} \tag{4.6}$$

where: (1) The division by 14 accounts for the fact that the dataset covers a 14-day period. (2) Multiplication by 365 scales it to an annual retrieval rate. (3) Division by 12 normalizes the retrieval rate per month. This methodology provides an accurate estimate of how often stored data is retrieved across warehouses.

Snowset Retrieval Rate Result. We executed this query on the snowset dataset, which yields a retrieval rate of `4.842`. This mean that for each stored GB, we need to be able to retieve it 4.842 times per month to meet realistic throughput requirements.

Approximating Query Selectivity Distribution. We aim to estimate the distribution of incoming queries over time by analyzing their selectivity, which measures the proportion of the warehouse data scanned by each query. For each query i, we define the `query selectivity` as:

$$S_i = \frac{q_i}{\hat{W}_i} \tag{4.7}$$

where: (1) q_i represents the total data read by query i (i.e., the number of scanned bytes). (2) $\hat{W}_i$ is the estimated warehouse size corresponding to query i. A higher selectivity value S_i indicates that the query scans a larger proportion of the warehouse, whereas lower values suggest more selective queries. Using the previously estimated warehouse sizes, we compute the selectivity ratio for each query as:

$$S_i = \begin{cases} \frac{q_i}{\hat{W}_i}, & \text{if } \hat{W}_i > 0 \\ \text{NULL}, & \text{otherwise} \end{cases} \tag{4.8}$$

This ensures that we avoid division by zero when estimating selectivity.

Binning Query Selectivity into Buckets. To analyze the distribution, we discretize the continuous selectivity values into percentage-based buckets:

$$B_i = \min\left(\lceil S_i \times 100 \rceil, 100\right) \tag{4.9}$$

where: (1) B_i is the `selectivity bucket` for query i, rounded up to the nearest whole number. (2) The `min` function ensures that values exceeding 100% are capped at 100%.

Normalized Selectivity Distribution. To compute the `probability mass function (PMF)` of query selectivity, we define n_p as the number of queries that fall into a specific selectivity percentage bin P_p:

$$n_p = \sum_{i=1}^{N} \mathbb{1}(B_i = P_p) \tag{4.10}$$

where: (1) $\mathbb{1}(\cdot)$ is an indicator function that is 1 if B_i belongs to percentage bin P_p, and 0 otherwise. (2) N is the total number of queries. The `normalized probability mass function (PMF)`, which expresses the `percentage of queries in each selectivity bucket`, is then:

$$P(p) = \frac{n_p}{N} \times 100 \tag{4.11}$$

where: (1) $P(p)$ represents the `percentage of queries` falling into selectivity bucket p. (2) n_p is the `number of queries` in bucket p. (3) N is the `total number of queries`, ensuring proper normalization.

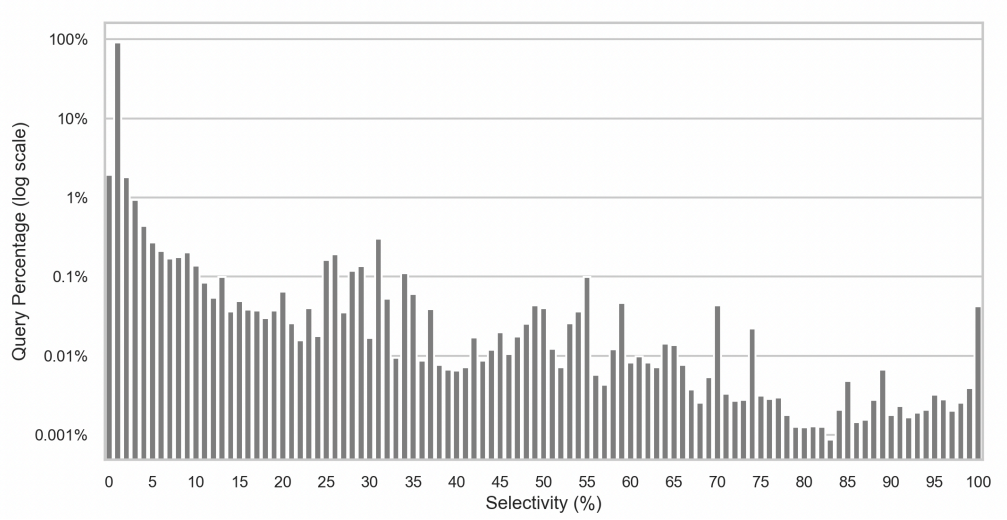

Figure 4.1: Snowset Selectivity Distribution

Result Discussion. The output of the SQL query provides a `normalized histogram of query selectivity percentages`, illustrating the distribution of queries across different levels of warehouse data scanned. This allows us to analyze the proportion of queries that access only a small fraction of the warehouse versus those that perform full or near-full scans. From the results, we observe that the vast majority (91.11%) of queries fall within the (0, 1] percentage range, meaning they scan at most 1% of the warehouse size. This indicates that most queries are highly selective, retrieving only small portions of the stored data. Additionally, 1.93% of queries have a selectivity value

Table 4.2: Snowset: Query Selectivity Distribution

Selectivity Range (%)	**Percentage of Queries**
0	1.93%
(0; 1]	91.11%
(1; 2]	1.81%
(2; 3]	0.93%
(3; 4]	0.44%
(4; 5]	0.27%
(5; 100]	3,51%

of exactly 0%, meaning they did not scan any data despite being issued. These queries may correspond to metadata lookups, failed executions, or optimizations that prevent unnecessary data scans. The remaining 6.96% of queries are distributed across higher selectivity ranges, with only a small fraction (3,78%) of queries scanning more than 5% of the warehouse. This suggests that full warehouse scans are rare, reinforcing the effectiveness of query pruning mechanisms. These findings highlight the predominance of highly selective queries, which are likely optimized for efficiency and reduced I/O overhead. The low number of queries with high selectivity suggests that full warehouse scans are minimized, potentially due to indexing strategies, partition pruning, or other query optimization techniques.

4.2 The Thesios Dataset

For our second analysis, we examine the Thesios dataset, a synthetically generated collection of I/O traces [19]. The original authors describe their synthetic trace generation methodology, demonstrating that their approach achieves high fidelity in reconstructing I/O patterns from down sampled real-world traces [20]. These traces capture READ and WRITE operations across various Google Cloud storage platforms, including Spanner, Bigtable, and Blobstore. The dataset spans 2 months from January 15th 2024 to March 15th 2024 and, similar to the Snowset dataset, contains sufficient information to allow for: (1) The estimation of the monthly retrieval rate, i.e., how frequently stored data is accessed. (2) The computation of the selectivity distribution for queries within the traces. To better understand the Thesios dataset and its implications for query aggregation, we first analyze its underlying data structure and observed anomalies.

Preprocessing: Filtering Non-Disk Reads. Since we are primarily interested in disk throughput, we filter out all requests that are: (1) Served from flash cache (Google's

internal spilling SSDs). (2) Cache hits (requests that are resolved entirely in memory). This ensures that our analysis focuses solely on I/O operations executed at the disk level.

Observations on I/O Trace Anomalies. To estimate the query size for a given trace, we analyze the following key attributes: (1) For WRITE requests, we expect: `request_io_size_bytes` = `disk_io_size_bytes` and confirm that this relationship holds in all observed cases. (2) For READ requests, we would expect: `response_io_size_bytes` = `disk_io_size_bytes`
However, the observed data exhibits an unexpected pattern: `request_io_size_bytes` = `response_io_size_bytes` and `disk_io_size_bytes` deviates by: (1) $1\% - 8\%$ lower than `response_io_size_bytes` in most cases. (2) $1\% - 4\%$ higher in some exceptional cases. The cause of these inconsistencies remains unclear, but potential explanations could include: (1) Compression or deduplication mechanisms affecting disk-level reads. (2) Additional read-ahead buffering strategies in the storage infrastructure. (3) Metadata overhead leading to variations in physical I/O sizes. To ensure robust analysis, we must adjust our computations to accommodate these deviations. Therefore we will only use request and response size in the following sections.

Table 4.3: Relevant Thesios Trace Attributes

Column	Description
filename	File identifier associated with the trace operation
file_offset	Byte offset at which the read or write operation begins
op_type	Type of operation: READ or WRITE
request_io_size_bytes	Size of the initial request in bytes
response_io_size_bytes	Size of the actual response in bytes (for reads)
disk_io_size_bytes	Actual number of bytes read from / written to disk
from_flash_cache	Indicator for whether request was served from flash cache
cache_hit	Indicator for whether request was fulfilled from memory cache

Estimating File Sizes. To analyze data access patterns within Thesios, we estimate the file sizes using observed trace executions. The data scanned per trace can be indirectly estimated using the `request_io_size_bytes` and `reponse_io_size_bytes` attributes:

(1) For WRITE operations, the file size estimate is:
$F_w =$ `file_offset` + `request_io_size_bytes`.

(2) For READ operations, the estimated file size is:

$$F_r = \texttt{file\_offset} + \texttt{response\_io\_size\_bytes}.$$

Aggregating File Size Estimates. To obtain a global estimate of file sizes over time, we: (1) Aggregate all traces for a given time window. (2) Compute the maximum observed file size across all traces.

Iterative Procedure for File Size Estimation. Since the dataset is available in small downloadable chunks, we iteratively refine file size estimates by: (1) Maintaining a running estimate for each observed file. (2) For each trace, computing a new file size estimate and updating it as: $F_{\text{updated}} = \max(F_{\text{current}}, F_{\text{new}})$ (3) Excluding any traces served from flash cache or memory cache.

Selectivity Estimation. For each read trace, we define query selectivity as:

$$S_i = \frac{\texttt{request\_io\_size\_bytes}}{F_i} \tag{4.12}$$

where: (1) S_i is the selectivity ratio for trace i. (2) F_i is the estimated file size at the time of the trace. We incrementally count occurrences of each selectivity value, constructing a selectivity distribution like with the snowset dataset.

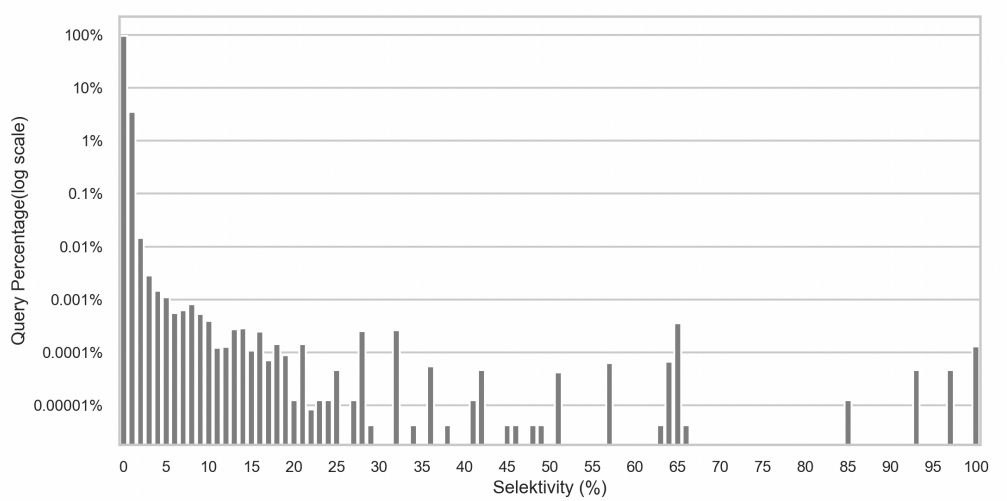

Figure 4.2: Thesios Selectivity Distribution

Result Discussion. Figure 4.2 highlights the selectivity distribution. We see a heavily skewed selectivity distribution towards 0%, showing a stronger bias than in the snowset dataset. We also notice missing selectivity distributions. We think this is due to two reasons: (1) data beyond 20% selectivity represent outliers only containing a single digit amount of recorded requests. These requests could be repititions of the same

Table 4.4: Thesios: Query Selectivity Distribution

Selectivity Range (%)	**Percentage of Queries**
[0; 1]	99.99%
(1; 100]	0.01%

underlying query. (2) the dataset itself is synthesized. We therefore expect regions of selectivity to not be sampled during dataset creation. We see a 99.99 % of queries having a selectivity below 0% and only 0.01% of queries having a selectivity above that.

Throughput Estimation. To compute the disk throughput requirement, we sum all bytes read across traces and divide by the aggregated file sizes:

$$T = \frac{\sum_{i \in R} \texttt{request\_io\_size\_bytes}}{\sum_{j \in F} F_j} \tag{4.13}$$

where: (1) T represents the average retrieval rate (number of times stored data is read). (2) R is the set of all read traces. (3) F is the set of all observed files. Since the dataset is evaluated for two months, we divide the retrieval rate by two:

$$T_{\text{monthly}} = T \div 2 \tag{4.14}$$

The computed result is $T_{\text{monthly}} = 17.84$, meaning that, on average, each stored byte is retrieved 17.84 times per month.

4.3 Conclusion.

Our analysis of the Snowset and Thesios datasets has yielded important insights into real-world query behavior and the corresponding throughput requirements of decentralized object storage systems. The Snowset dataset reveals that an overwhelming majority of queries are highly selective, with approximately 91.11% of queries scanning at most 1% of the respective warehouse dataresulting in an average retrieval rate of 4.842 per stored GB per month. In contrast, the Thesios dataset, despite its synthetic nature, indicates a retrieval rate of $T_{\text{monthly}} = 17.84$, reinforcing the notion that even when data is accessed frequently, the fraction of data read remains very low. These findings underscore the potential benefits of near-storage pre-filtering, as filtering a small, relevant subset of data can substantially reduce unnecessary disk I/O and improve overall throughput. The insights gained here lay the groundwork for assessing whether our EC2-based decentralized object storage system meets real-world performance demands.

5 Architecture

Motivation and Outline. In this chapter, we propose a *decentralized object storage service* architecture that supports near-storage query execution. Our design addresses two primary problems found in established implementations: (1) the bottleneck at a central metadata node, and (2) excessive disk and network usage when retrieving and filtering large data sets. In this paper we will focus on querying parquet files, as they allow us to store metadata such as min and max values for each column and subset of rows called row group. These metadata help us skipping over row groups while reading, reducing I/O and increasing throughput in the process. We begin by describing the traditional architecture that highlights these issues, then introduce our improved Metadata-Storage Node design, showing how it allows storage nodes to prune unnecessary data before sending results.

5.1 State-of-Art Architecture

Assume a collection of *storage nodes*, each responsible for storing portions (partitions) of various objects (files). A separate *metadata node* tracks where each partition is located. When a file is uploaded, it is split into smaller pieces, which are placed on different storage nodes.

Conventional Workflow. In the conventional design, a client who wants to query data, proceeds as follows: (1) The client sends the query to the metadata node. (2) The metadata node determines which storage nodes hold the data of interest. (3) The metadata node requests the relevant partitions from each storage node. (4) Each storage node responds by returning the *entire data* stored by this specific user to the metadata node. (5) The metadata node (in this case acting also as the the compute node) combines this data, applies the clients filter or predicate, and sends the final filtered data back to the client. While simple to implement, this approach has two key drawbacks: (1) *Metadata Node Bottleneck.* All query traffic flows through one node, limiting scalability and risking saturation of network/compute resources at the metadata node. (2) *Excessive I/O and Network Transfer.* Even if the query is highly selective (as discussed in the previous chapter), the system still ships full data partitions across the network before filtering.

5.2 Proposed Metadata-Storage Node Architecture.

Our design overcomes the above limitations with two main changes: (1) *Direct Client–Storage-Node Communication:* Instead of routing large data partitions through the metadata node, the client contacts storage nodes directly once it received information from the metadata node on which nodes the needed data is stored. (2) *Near-Storage Compute:* We will store data in parquet format. This allows storage nodes to partially execute queries by skipping rows or columns that are guaranteed not to match. As a result, less data is read from disk and sent over the network.

Metadata Node. The metadata node still manages partitioning and stores location information for each file. On an `UPLOAD` operation: (1) *Erasure Coding or Replication:* Creates redundant copies or parity shards for fault tolerance. (2) *Partitioning:* Splits the file into multiple partitions and distribute them across different storage nodes. (3) *Metadata Registration:* Records which node holds each partition. When the client sends a `QUERY`, the metadata node simply returns the list of storage nodes needed for the client to locate and access the relevant data.

Storage Node. Each storage node stores partitions locally on disk and has enough compute resources (CPU and RAM) as shown in chapter 3.2 to filter out irrelevant data before sending results. This process typically involves: (1) *Reading and Storing File Metadata:* Upon upload, the metadata is deserialized and stored in memory for quick query access. (2) *Row-Group Pruning:* After receiving a query, the metadata is checked for min and max values for each row group. If the query condition (e.g., `Age` > 10) cannot possibly be satisfied by a particular row group, the node skips reading that group entirely. (3) *Selective Reading:* For row groups that pass the pruning check, the node can further skip columns that are not needed by the query. (4) *Lightweight Filtering in Memory:* As the node reads data from disk, it applies the filtering condition and immediately discards non-matching rows, reducing the amount of information that must transferred over the network.

End-to-End Query Execution Flow. The overall query process is: (1) The client requests metadata from the metadata node, which nodes hold each partition. (2) The client sends the query directly to the relevant storage nodes. (3) Each storage node locally applies row-group pruning, reads the necessary data from disk, does additional filtering, then sends only the matching results back to the client. (4) The client merges the results obtained from all contacted storage nodes. This architecture minimizes load on the metadata node and drastically reduces the volume of data read and transferred between storage nodes and the client, particularly when queries are selective.

5.3 Conclusion

The proposed design solves the scaling issues in a conventional architecture by allowing clients to communicate directly with storage nodes and by applying near-storage computation to avoid reading or transferring unnecessary data. This approach is especially beneficial for queries that filter out large portions of a dataset. Subsequent chapters will examine the storage nodes near storage compute implementation details and evaluate its performance in real-world scenarios.

6 Metadata-Driven Pruning

Outline and Motivation. In this chapter, we introduce our initial implementation for performing *near-storage computation* on Parquet files. The primary objective is to reduce the volume of data read from disk and transferred over the network by skipping irrelevant row groups and columns before reading. This approach builds on top of the `arrow2` and `parquet2` crates in Rust, which provide higher-level abstractions for Parquet data reading and Arrow-based columnar processing. We introduce four main features in this first approach: (1) *Row Group Pruning*: Excluding row groups based on Parquet *min* and *max* statistics at read time. (2) *Row Filtering*: Discarding individual rows according to a user-specified predicate. (3) *Aggregation*: Computing aggregates (e.g., min, max, $\sum$, count, avg) as rows are read. (4) *Column Projection*: Only reading the subset of columns that is actually needed for the query and are specified by the user. The code listing 6 demonstrates our approach, in which we parse user queries into an *expression tree* and introduce the concepts of *row-group pruning at read time*, *row-level filtering*, *aggregation*, and *column pruning* (projection) to reduce I/O. By leveraging Parquet metadata to skip entire row groups, each of which contains multiple column chunks, that cannot match, and by further omitting unneeded column chunks within those row groups, we significantly reduce the amount of data accessed. Nonetheless as we show in our evaluation 8, the overhead of *metadata parsing*, *decompression*, and *deserialization* in `arrow2/parquet2` remains considerable, especially in scenarios with many row groups and low query selectivity.

6.1 Row Group Pruning

A Parquet file is split into multiple *row groups*, each of which stores basic statistics (e.g., the minimum and maximum) for every column it contains. Suppose a user query includes a predicate φ. Our goal is to *skip* an entire row group g if, solely from gs min/max statistics, we can deduce that φ cannot possibly be satisfied by any row in g. Formally, define:

$$\text{keep}(g, \varphi) \;=\; \begin{cases} \text{true}, & \text{if } \varphi \text{ may hold for at least one row in } g, \\ \text{false}, & \text{if } \varphi \text{ is guaranteed not to hold for any row in } g. \end{cases}$$

We *prune* (skip) row group g if $\text{keep}(g, \varphi) = \text{false}$.

```
1: function ExecuteQuery(fileMeta, φ_string, α_string, columns)
2:     φ ← parseExpression(φ_string)
3:     α ← parseAggregation(α_string)
4:     columns_needed ← columns ⊕ α.columns ⊕ φ.columns
5:     RG_pruned ← [ ]
6:     for each row group g in fileMeta.allRowGroups do
7:         if KeepRowGroup(g, φ) = true then
8:             RG_pruned.append(g)
9:         end if
10:    end for
11:    aggregation ← EmptyTable
12:    for each row group g ∈ RG_pruned do
13:        dataBatches ← ReadColumns(g, columns_needed)
14:        for each chunk batch in dataBatches do
15:            M ← BuildFilterMask(batch, φ)
16:            filteredBatch ← ApplyMask(batch, M)
17:            filteredBatch ← FilterColumns(columns)
18:            aggregation ← aggregation ⊕ Aggregate(filteredBatch)
19:            streamToClient(filteredBatch)
20:        end for
21:    end for
22:    streamToClient(aggregation)
23: end function
```

Handling Complex Predicates. A predicate φ can be an expression over multiple columns, potentially combining conditions with $\wedge$ (`AND`), $\vee$ (`OR`), and $\neg$ (`NOT`). Rather than evaluating φ *column by column*, we walk the expression tree for φ. At each *leaf* of the tree, there is a simple condition on a single column (e.g., Age $>$ 42 or City $=$ `"Munich"`). We check the row groups min/max statistics for that column to see if those values *definitely cannot* satisfy the leaf condition. The outcomes of these leaf checks are then combined according to φs logical operators.

Definite Skips. We conclude keep$(g, \varphi) =$ false as soon as the algorithm deems φ *definitely impossible* for all rows in g. Hitting even one leaf condition that is guaranteed to fail can skip the row group for $\wedge$-chains. For $\vee$-chains, all branches must fail under that same context. This short-circuit behavior prunes row groups *without decompression and deserialization*, often saving significant I/O and CPU cycles when many groups do not satisfy the query.

6.2 Row Filtering

Although row group pruning can eliminate large chunks of data, many row groups still contain partially relevant rows. We thus introduce *row filtering*, which discards rows *as* they are read from disk and deserialized by `arrow2`. Formally, for each batch of rows loaded from a row group, we create a Boolean *mask array*:

$$M[i] = \begin{cases} \text{true}, & \text{if row } i \text{ satisfies } \varphi, \\ \text{false}, & \text{otherwise.} \end{cases}$$

We then keep only those elements of each columns array corresponding to true entries in M. This is implemented via the Arrow compute kernel filter_chunk$(\cdot)$: (1) *Compute a mask for each chunk:* We traverse the expression tree φ (which may contain $\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}$) and evaluate it row-by-row to build a Boolean mask M. (2) *Apply the mask:* We call filter_chunk$(batch, M)$, producing a new chunk containing only rows with true entries. (3) *Discard filtered rows:* The unwanted rows are dropped in-memory, never returned to the user. Because row filtering cannot skip bytes on disk (the entire row group data is read and deserialized anyway), it incurs additional CPU overhead from the compare operations.

6.3 Aggregation

In many analytical workloads, the user is interested in aggregates such as min, max, count, sum, or avg over (possibly filtered) rows. Our approach allows computing these aggregates after the row filtering step. Conceptually, we can update running counters and partial aggregates as each filtered batch arrives. Let C denote the column of interest, and suppose we wish to compute

$$\min(C), \quad \max(C), \quad \sum(C), \quad \text{count}(C), \quad \text{and} \quad \text{avg}(C).$$

After row filtering discards rows that do not satisfy φ, each remaining row in the chunk for column C contributes to the aggregator. For instance, we update:

$$\text{count} \leftarrow \text{count} + (\text{number of surviving rows}),$$

$$\sum \leftarrow \sum + \sum_{r \in chunk} C[r].$$

At the end, $\text{avg}(C) = \sum / \text{count}$. Similar to row filtering, this only incures additional CPU overhead from the aggregate operations, as well as negligible memory overhead for storing results.

6.4 Column Projection

A further optimization is *column projection*, which avoids reading unneeded columns from the Parquet file. Conceptually, if the user query references columns $\{C_1, \ldots, C_k\}$,

then we skip columns $\{C_{k+1}, \dots\}$ entirely. Parquet naturally supports *columnar reads*, so the `arrow2` crate can be configured to read only a subset of columns in each row group: (1) *Determine columns needed for the predicate:* We analyze φ to identify the columns used in any row-group pruning or row-level filtering. (2) *Determine columns needed for the aggregation:* If the query includes an aggregation, we include these columns as well. (3) *Determine columns specified:* Finally, columns specified by the query itself are included. (4) *Read only these columns:* During the `FileReader::new` call, we specify a projection array indicating which columns to load. (5) *Filter Result:* After each row group and row filter batch has concluded, we discard any column not specified by the query itself. In practice, this yields significant I/O and CPU savings for wide tables where only a few columns must be read.

Implementation Algorithm 6 gives a high-level overview of how we combine row group pruning, row filtering, aggregation, and column projection. In our implementation, lines 2–3 parse the expression and aggregation. Lines 5–10 handle `row group pruning`. Lines 11–21 show how each row group is read in columnar form with only `projected columns`. Within each batch, lines 15–17 apply `row filtering` to discard unwanted rows, after which line 18 computes `aggregates` on the final subset. Lines 19 and 22 then stream the results to the client.

6.5 Challenges and Conclusion

Although we successfully reduce the volume of data processed, several factors can still lead to overhead when reading moderately large volumes of Parquet data: (1) *Metadata Deserialization Overhead.* Even though row group pruning looks only at min/max statistics, the `arrow2/parquet2` crates fully deserialize Parquet metadata. This step can consume significant CPU and memory, especially for files with many row groups. (2) *Decompression and Deserialization.* The entire row group (or at least the relevant columns) undergo decompression and Arrow deserialization. This overhead is especially high if the query selectivity is low and hardly any row groups can be skipped. (3) *Internal Copying and Buffering.* Certain operations in `arrow2` and `parquet2` make additional copies of data to conform to Arrows in-memory format, adding more CPU cost. (4) *Memory Footprint.* Large in-memory metadata can occupy gigabytes of RAM if we store many files' metadata simultaneously. These bottlenecks keep the system in some cases with small row groups and low selectivity from fully exploiting the raw disk bandwidth. In summary, this first approach lays essential groundwork for near-storage computation in a decentralized environment: (1) `Row Group Pruning`: Leverages Parquet min/max statistics to avoid reading entire row groups. (2) `Row Filtering`: Discards unnecessary rows in memory. (3) `Aggregation and Projection`: Minimizes columns read and computes aggregates on the fly. Nevertheless, as will be shown in the evaluation 8, metadata deserialization at the start of each query lead to CPU overheads that hinder peak performance. In the following chapter, we propose a *second approach* that circum-

vents much of the standard Parquet crates metadata parsing and deserialization. By storing compact *custom metadata* in-memory and reading only the exact required byte ranges from disk, we only need to deserialize the metadata once at file upload, we reduce CPU usage and memory footprint, thereby allowing near-native disk throughput.

7 Direct Byte-Range Access

Motivation. In chapter 6, we introduced an approach that harnessed row-group skipping as well as column projections to reduce the data scanned for query processing. This method leveraged the `parquet2` crate for Parquet metadata deserialization and the `arrow2` crate for row-wise data reading, permitting custom row group skipping, row filtering, aggregation, and column projection. However, our measurements indicate that this approach cannot use the instance memory effectively to store the files deserialized metadata, which results in an extra metadata deserialization step. Also the arrow reader is designed for random disk access, which degrades our disk performance. In this chapter we present our second approach which (1) avoids deserializing the metadata for each query as well as bypassing decompressing and deserializing the parquet file, saving CPU cycles (2) stores the files' metadata in-memory saving a disk roundtrip fetching and deserialzing the metadata (3) reads the row groups sequentially achieving higher disk throughput.

Challenges. Two central issues arose when using the default metadata reading and deserialization mechanisms: (1) *Excessive Memory Usage:* The deserialized Parquet metadata (e.g., row group statistics and column schemas) expands significantly in memory, consuming a lot of memory. Given that each node is constrained to 64 GiB of RAM, this overhead can become prohibitive when scaling to filling all available storage space. (2) *CPU Bottlenecks:* Despite optimizations (e.g., reducing encoding overhead), the process of deserializing Parquet files remained CPU-intensive. For low row group sizes and low selectivity, the CPU becomes a bottleneck, limiting how quickly data can be read from disk.

Solution Overview. To overcome these limitations, we adopt an alternative, *Direct Byte-Range Access* approach that bypasses metadata and data deserialization. Specifically, we store minimal metadata structures in memory and implement our own row-group pruning logic. This design achieves two major benefits: 1. *Reduced Memory Footprint:* By storing only essential information (e.g., file offsets, lengths, and statistics), we drastically cut down on the memory required for metadata. 2. *Direct Disk Access:* For each query, we compute the exact byte ranges needed from disk, enabling partial I/O and allowing the system to sustain high disk throughput.

7.1 Metadata Pruning

The core of this approach relies on a compact representation of column statistics. Listing 7.1 demonstrates the primary data structures: 1. *FileMeta*: Stores the number of columns (`columns`), row groups (`row_groups`), a list of column types (`col_types`), an array of per-column-chunk metadata (`col_meta`), and the position of the raw Parquet metadata footer (`meta_offset`, `meta_length`). 2. *ColumnChunkMeta*: Encapsulates the file offset, length, and minimum/maximum values (`min`, `max`) for each column chunk. 3. *ColumnType*: An enumeration for the physical column type (e.g., `Int32`, `Double`, etc.).

```
struct ColumnChunkMeta {
    offset: u32,
    length: u32,
    min: u64,
    max: u64,
}

struct FileMeta {
    columns: u32,
    row_groups: u32,
    col_types: Vec<ColumnType>,
    col_meta: Vec<ColumnChunkMeta>,
    meta_offset: u32,
    meta_length: u32,
}

enum ColumnType {
    Boolean,
    Int32,
    Int64,
    Int96,
    Float,
    Double,
    ByteArray,
}
```

Listing 7.1: Compact Parquet Metadata Structures

Memory Footprint. We derive the theoretical memory consumption of storing the metadata for a single Parquet file. Let C be the number of columns and G be the number of row groups.

$$M_{\text{file}} = M_{\text{fixed}} + (C \times G) \times M_{\text{chunk}} + C \times M_{\text{type}} \tag{7.1}$$

where: 1. M_{fixed} is the fixed cost (in bytes) for storing, for example, `columns`, `row_groups`, `meta_offset`, `meta_length`, and two `Vec` pointers. This sums up to 64 bytes. 2. $M_{\text{chunk}} =$ 24 bytes for each `ColumnChunkMeta`, storing `offset`, `length`, `min`, and `max`. 3. $M_{\text{type}} =$ 1 byte for the column type enumeration.

Scaling to Large Datasets. When managing tens of thousands of row groups per file, or thousands of files, the total in-memory footprint can grow significantly. However, because we no longer store fully deserialized Arrow metadata, the memory required remains manageable given careful partitioning of data (e.g., files under 4.29 GB to allow 32-bit offsets and lengths).

7.2 Optimizing Row Group Granularity

A smaller row group size increases the potential for row-group pruning (i.e., skipping irrelevant subsets of data), but also increases metadata size. Conversely, larger row groups reduce the memory overhead of metadata but limit pruning efficacy. We formalize the trade-off by balancing the number of row groups G and the memory budget B. We solve for the minimal row group size subject to memory constraints:

$$N \times \left(M_{\text{fixed}} + C \times M_{\text{type}} + G \times C \times M_{\text{chunk}}\right) \;\leq\; B.$$

Approximate Computation: Suppose each node has a memory budget B for metadata, then the maximum permissible row-group count G_{max} is given by:

$$G_{\text{max}} \approx \frac{B - N \times (M_{\text{fixed}} + C \times M_{\text{type}})}{N \times C \times M_{\text{chunk}}}, \tag{7.2}$$

where N is the number of files (i.e., how many times the dataset repeats on a node). The corresponding row group size in terms of number of rows results from dividing the total row count by G_{max}.

Example: Scaling the Snowset Dataset. A single Snowset file with *93* columns, *125,786* row groups, and *550* rows per row group requires roughly 14.352 GB of storage space. Now suppose each node in the cluster has: 1. *111,840* GB of total storage, and 2. *62* GB of available memory for storing metadata. Hence a single node can store

$$N \;=\; \left\lfloor \frac{\text{111,840 GB}}{\text{14.352 GB}} \right\rfloor \;=\; \text{7,792}$$

copies of the Snowset dataset. Substituting the concrete values:

$$B \;=\; 62\,\text{GB} \;=\; 6.2 \times 10^{10}\,\text{bytes}, \quad M_{\text{fixed}} \;=\; 157\,\text{B},$$

$$M_{\text{type}} \;=\; 1\,\text{B}, \quad N \;=\; \text{7,792}, \quad C \;=\; 93, \quad M_{\text{chunk}} \;=\; 24\,\text{B},$$

we obtain

$$G_{\text{max}} \;\approx\; \approx\; \text{27,777,229 row groups (total).}$$

Thus, each file can support

$$\frac{\text{27,777,229}}{\text{7,792}} \;\approx\; \text{3,565}$$

row groups per Snowset dataset instance under the 62 GB metadata limit.

Rows per Group. Given 69,182,074 rows in each Snowset dataset, the 3,565 permissible row groups imply a row group size of approximately

$$\text{rpg} \;=\; \frac{69{,}182{,}074}{3{,}564} \;\approx\; 19{,}406 \text{ rows per group.}$$

Note that a row group of $\approx 19{,}406$ rows represents a compromise between the *desire for small row groups* (for maximum pruning) and the *need to control memory* usage.

Result Discussion. By carefully balancing row group sizes, we can guarantee that the system respects its overall memory budget for metadata across thousands of files, while still retaining enough granularity to prune large portions of data at query time.

7.3 Implementation

Metadata Parsing and Encoding Upon file upload, we perform a single pass over the Parquet metadata to extract column offsets, lengths, and min/max statistics: (1) *Reading the Footer*: We seek to the end of the file to locate the Parquet magic bytes (`PAR1`) and read the metadata length. (2) *Deserializing Statistics*: We parse min and max values from the built-in Parquet statistics objects, encoding them in 64-bit format using custom routines (`encode_f64`, `encode_byte_array`, etc.). (3) *Storing Column Types*: Each column's physical type (e.g., `Double`, `ByteArray`) is recorded in a compact `enum`. (4) *Finalizing FileMeta*: We populate `FileMeta` with the row group and column information, discarding the original large in-memory Parquet metadata structure. This also lets us skip the metadata deserialization on each query.

Row Group Pruning Using Expression Trees Once the metadata is loaded, queries are formulated as logical expressions. For each row group, we check whether it can be skipped by comparing its stored min/max values against the query predicates. The pruning algorithm can be formalized as follows:

$$\text{keep}(g) = \bigwedge_{c \in C(g)} \Gamma\big(\min_c, \max_c, \text{expr}\big), \tag{7.3}$$

where g denotes a row group, $C(g)$ the set of columns in row group g, and Γ is a boolean function indicating whether the row group satisfies the query expression. If `keep`(g) is false, we exclude the corresponding byte range from the subsequent read. The algorithm to decide which row groups to keep, remains the same as in chapter 6.

Implementation. The code implements a recursive expression-evaluation function that decodes the stored metadata, checks it against the user threshold, and returns true or false. We then collect all surviving row groups and merge overlapping byte ranges before reading them from disk.

Conclusion. We present a memory-efficient as well as CPU-efficient and HDD-friendly method for row-group pruning in Parquet files. By storing only essential metadata (min/max statistics and column chunk offsets), we avoid the large overheads associated with full deserialization. Our approach: (1) Achieves near-native disk throughput in worst-case scans. (2) Scales effectively to large data volumes by minimizing in-memory metadata. (3) Enables extreme speed-ups in scenarios where row-group skipping excludes most data.

8 Evaluation

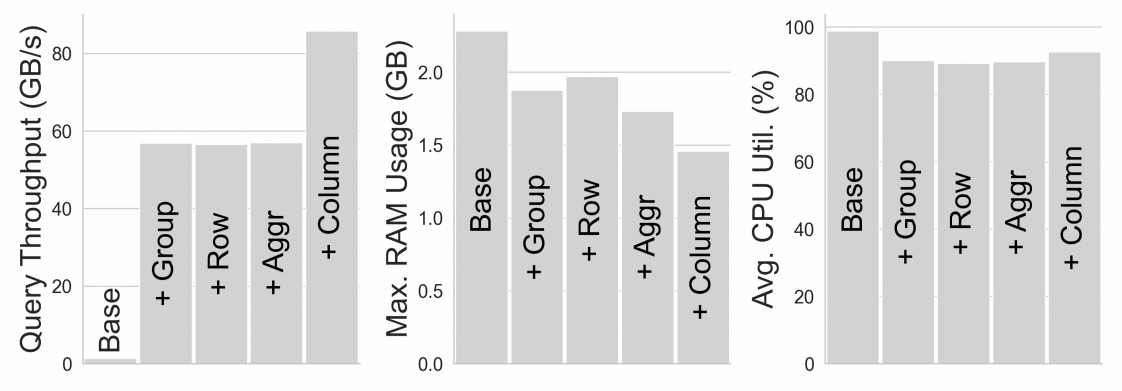

Figure 8.1: Query throughput, Max. RAM Usage and Avg. CPU utilization changes when successively enabling techniques

Outline. In this chapter, we present a comprehensive evaluation of our two approaches for near-storage query processing on Parquet files. Our investigation focuses on three key metrics: (1) *Query Throughput*, (2) *Disk Throughput*, (3) *RAM Usage*, and (4) *CPU Utilization*. Furthermore, we analyze the dependence of throughput on query selectivity and derive expected retrieval rates by integrating empirical selectivity distributions from real-world datasets as shown in chapter 4. This multi-faceted evaluation not only quantifies the performance improvements achieved by each feature but also highlights the trade-offs between the two approaches. To maintain brevity, we subsequently refer to the first approach as *Parquet Reader* or *V1* and the second approach as *Direct Byte Access* or *V2*.

Test Setup. All experiments are executed on an AWS `d3en.4xlarge` instance with the following specifications: (1) *CPU:* 16 cores, (2) *RAM:* 64 GB, and (3) *Disk:* Eight hard disks with a combined theoretical throughput of 2 GiB/s (measured at approximately 1.978 GiB/s in chapter 3.2). To simulate realistic production conditions, we query the Snowset dataset. In order to avoid caching effects and achieve a steady state, the dataset is copied 40 times. For each query, query throughput is calculated as:

$$\text{Query Throughput} = \frac{\text{Dataset Size Scanned}}{\text{Total Query Execution Time}}.$$

Rationale for Local Throughput Testing. Although this system is designed for distributed storage on EC2, our microbenchmarks (see chapter 3.2) have shown that the process is disk-bound rather than network-bound. Consequently, network throughput does not significantly impact overall query performance, and we therefore focus on disk throughput. In addition, sending data over the network incurs only moderate overhead: approximately 7.98% CPU usage and 501 MB of memory for synchronization and data transfer. These levels are insufficient to render the system CPU-bound or to exhaust available memory resources. We therefore conduct all throughput tests locally while still accounting for the additional 501 MB of memory overhead in our memory calculations.

8.1 Impact of Individual Technique Enhancements.

Our Parquet Reader approach introduces four key enhancements: row group pruning, row filtering, aggregation, and column projection. We first assess the performance impact of successively enabling individual features. The features are evaluated in terms of their effect on query throughput, memory consumption, and CPU load.

Throughput Improvements. The key observation is that the number of row groups and columns skipped is directly related to the query throughput. When irrelevant row groups are pruned, less data is read and deserialized. We observe: (1) *Row Group Filtering:* Increases throughput from a base of 1.624 GB/s to 56.801 GB/s, an improvement of approximately 3498%. (2) *Column Filtering:* Further enhances throughput to 85.768 GB/s (an additional 150% improvement over row group filtering). Overall, the combination of row group and column filtering yields an improvement of around 5281% for queries with a 1% row group and column selectivity.

Memory Usage Reduction. Memory usage is measured relative to the systems baseline consumption, accounting for metadata storage of 640 Parquet files (each with 223 row groups and 19,406 rows per group). Notably: (1) Enabling row group filtering decreases memory usage from 2.337 GB to 1.920 GB, which is an 18% reduction. (2) With the final configuration (after column filtering), overall RAM consumption is reduced by 845 MB, corresponding to a 36% decrease.

CPU Utilization. CPU utilization is averaged across all 16 cores. Our measurements indicate: (1) *Row Group Filtering* alone reduces CPU load from 98.81% to 90.11% (a reduction of approximately 9%) by avoiding unnecessary decompression and deserialization. (2) *Row Filtering and Aggregation* introduce only marginal CPU overhead (e.g., aggregation increases CPU usage by merely 0.4%). (3) *Column Filtering* yields a final CPU utilization of 92.62%, which represents an overall improvement of about 6% compared to the base case.

Table 8.1: Query Throughput

Selectivity	*Throughput (GB/s)*	
	Parquet Reader (Group Feature)	*Direct Byte Access*
100% (Worst Case)	1.624	1.715
50%	1.661	1.709
25%	1.707	1.805
10%	3.151	3.354
1% (Realistic)	56.801	60.738
0% (Best Case)	112.511	1655.853

8.2 Throughput as a Function of Query Selectivity

Outline and Motivation. To further explore the relationship between query selectivity and throughput, we measured performance for a range of selectivity values from 0% (best-case, where no row group matches the query) to 100% (worst-case, where every row group is processed). Table 8.1 summarizes these measurements for both approaches . In this chapter, we leverage these results to derive an expected retrieval rate for both approaches. In chapter 4, we examined retrieval rates from real–world distributed storage systems, namely Snowflake and Google Storage, establishing a realistic requirement to retrieve stored data between 4.842 and 17.84 times per month. To obtain an estimate of throughput across different levels of query selectivity, we fit an approximate function to our data points and then use the query distribution (also determined in chapter 4) to compute the expected retrieval rate. These findings allow us to determine whether the throughput achieved by our approaches is sufficient for practical, real-world scenarios.

Theoretical Retrieval Rate. Prior to incorporating near-storage computation and transmitting entire files to the client, we derive a baseline monthly retrieval rate using the theoretical disk throughput of 2 GiB/s (2.147484 GB/s). The resulting monthly retrieval rate R_{disk} can be approximated as:

$$R_{\text{disk}} \;=\; \frac{365 \times 24 \times 3600}{\frac{111840\,\text{GB}}{2.147484\,\text{GB/s}}} \;\approx\; 50.46,$$

which already exceeds the target retrieval rate requirements. In what follows, however, we assess the additional benefits that both of our near-storage approaches provide in improving retrieval rates.

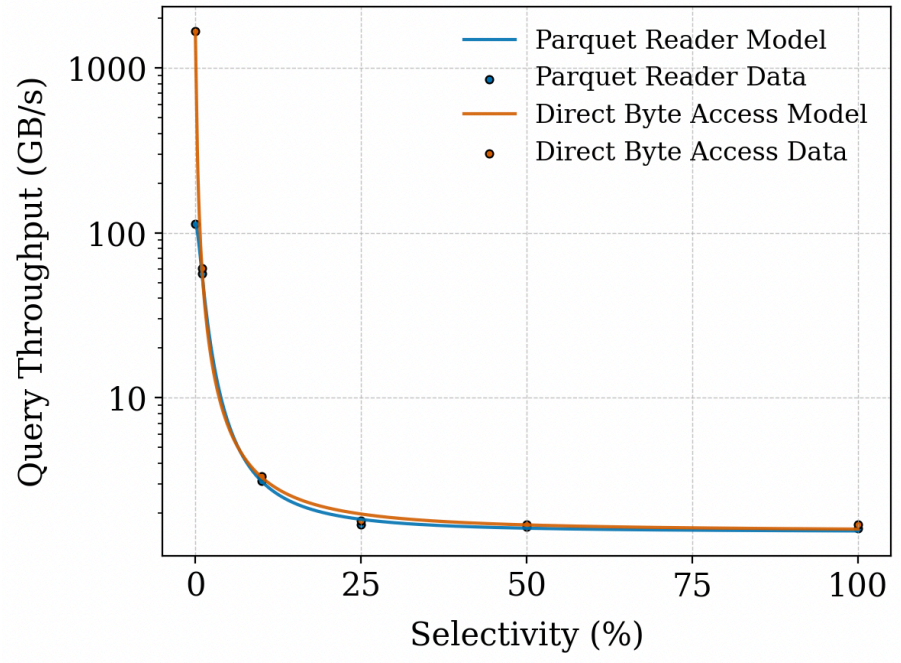

Figure 8.2: Query throughput per workload selectivity

Regression. Table 8.1 reveals an exponential increase in throughput T as the query selectivity x decreases. This relationship can be modeled by a rational function:

$$T(x) = a + \frac{b}{1 + c\,x^d},$$

where the parameters a, b, c, and d are estimated using regression techniques (e.g., via the `scipy.optimize.curve_fit` function). For instance, the fitted models for the two approaches are:

$$T_{\text{V1}}(x) = 1.5386 + \frac{110.9723}{1 + 1.0081\,x^{1.8359}}, \quad T_{\text{V2}}(x) = 1.5528 + \frac{1654.3002}{1 + 26.951\,x^{1.5445}}.$$

Using the fitted throughput function $T(x)$ and the observed probability mass function $P(x)$ of query selectivities in chapter 4, the expected throughput $E[T]$ is computed as:

$$E[T] = \sum_{x=0}^{100} T(x) \cdot \frac{P(x)}{100}.$$

Result Discussion. From this model, we derive the monthly retrieval rate for each approach. For our evaluated approaches, the retrieval rates are summarized in Table 8.2. These values indicate that both approaches surpass the retrieval rate requirements observed in real-world scenarios (4.842 and 17.84 retrievals per GB per month for Snowset

Table 8.2: Query Throughput and Retrieval Rates

Selectivity Distribution	*Query Throughput (GB/s)*		*Retrieval Rate per month*	
	V1 (Group Feature)	*V2*	*V1 (Group Feature)*	*V2*
Snowset	54.673	88.011	1284.70	2068.08
Thesios	56.791	60.727	1334.46	1426.95

Table 8.3: Relative Error.

Approach	**Select. (%)**	**Measured (GB/s)**	**Fitted (GB/s)**	**Rel. Error (%)**
V1	100	1.6240	1.5620	-3.82
V1	50	1.6610	1.6222	-2.34
V1	25	1.7070	1.8365	7.59
V1	10	3.1510	3.1219	-0.92
V1	1	56.8010	56.8014	0.00
V1	0	112.5110	112.5109	0.00
V2	100	1.7150	1.6028	-6.54
V2	50	1.7090	1.6986	-0.61
V2	25	1.8050	1.9782	9.60
V2	10	3.3540	3.3030	-1.52
V2	1	60.7380	60.7384	0.00
V2	0	1655.8530	1655.8530	0.00

and Thesios, respectively). Figure 8.2 highlights the exponential relationship between query throughput and selectivity. Table 8.3 shows the relative errors achieved from our fitted functions.

8.3 Metric Comparison

To further contrast the two approaches, we conducted experiments at a fixed query selectivity of 10% and measured resource utilization.

CPU Utilization. For approach *Parquet Reader*, the CPU load ranges from approximately 75% to 90%, reflecting the overhead of repeated metadata deserialization and row group processing. In contrast, approach *Direct Byte-Range Access* demonstrates a lower CPU load (between 60% and 75%), due to its efficient in-memory handling of minimal metadata.

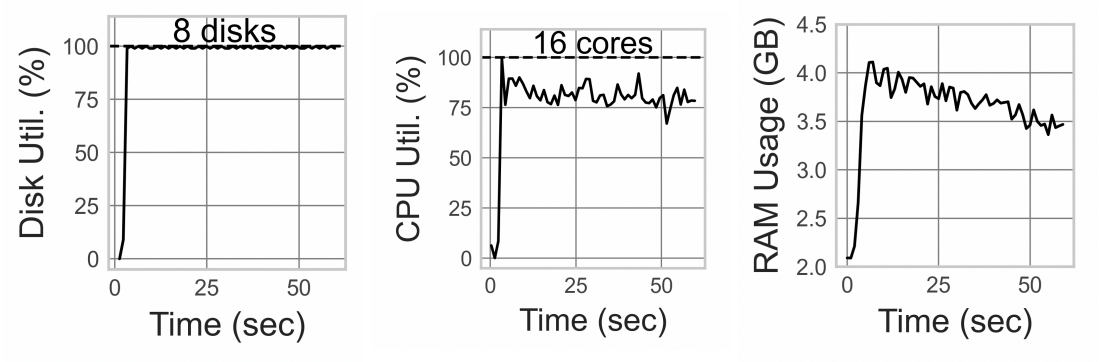

Figure 8.3: Approach 1 (Group feature) resource utilization for selectivity 10%

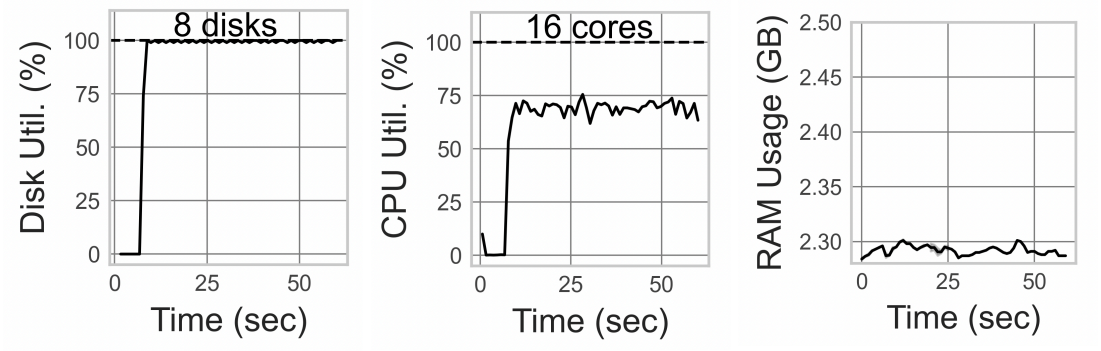

Figure 8.4: Approach 2 resource utilization for selectivity 10%

Memory Usage. The memory profile of the *Parquet Reader* approach is characterized by an initial spike due to the deserialization of raw metadata for 640 Parquet files; however, memory usage stabilizes after query execution, with a peak consumption of roughly 4.2 GB. In comparison, *Direct Byte-Range Access's* design, using preprocessed, minimal metadata structures, results in significantly lower additional memory consumption, as only small buffers are required for raw disk I/O.

8.4 Metadata Overhead

In this section, we evaluate the memory usage of both proposed approaches. We adopt a row group size of 19 406 rows per group, consistent with our previous experiments, such that our second approach remains theoretically within the memory budget described in chapter 7. For the first, *Parquet Reader* approach, we measure two components of memory usage: (i) the metadata in parsed form (i.e., the final data structure format employed by the parquet reader) and (ii) the raw byte form of the metadata footer located at the end of the file, which must be deserialized before it becomes usable. For our *Direct Byte-Range Access* approach, we also measure the in-memory footprint of

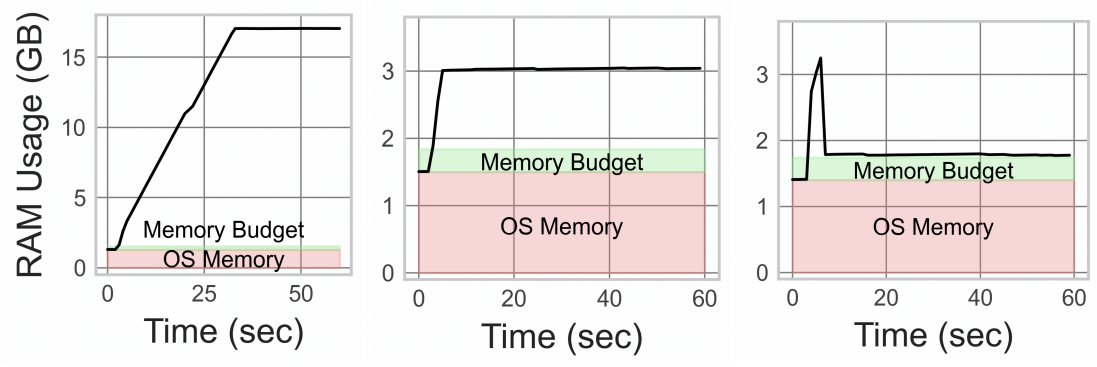

Figure 8.5: Parsed, raw and minimized struct metadata memory overhead

the metadata. All three measurements begin by reading the metadata footer. In the first measurement, we deserialize the metadata entirely and store it in memory. In the final measurement, we parse and convert the metadata into our minimized metadata structures.

Memory Budget. We derive the available memory budget for in-memory metadata storage as follows. Let $M_{\text{total}} = 64\,\text{GB}$, $M_{\text{system}} = 1.5\,\text{GB}$ and $M_{\text{program}} = 0.5\,\text{GB}$. Hence, the memory remaining for metadata is $M_{\text{budget}} = M_{\text{total}} - \left(M_{\text{system}} + M_{\text{program}}\right) = 64\,\text{GB} - 2\,\text{GB} = 62\,\text{GB}$. Next, assume a total storage capacity of $S_{\text{total}} = 111{,}840\,\text{GB}$. Then, for every gigabyte of storage, we can allocate $\frac{M_{\text{budget}}}{S_{\text{total}}} = \frac{62\,\text{GB}}{111{,}840\,\text{GB}} \approx 0.554\,\text{MB}$. In our experiment, we store a total of 640 Parquet files from the *snowset* dataset with a combined size of 432.576 GB. Thus, the memory budget allocated to these files is $432.576\,\text{GB} \times 0.554\,\text{MB/GB} \approx 0.240\,\text{MB}$.

Result Discussion. We observe that fully deserializing the metadata significantly exceeds the memory budget. The combined footprint of all deserialized metadata is approximately 15.73 GB, surpassing the budget by about 6454%. Even when storing only the raw footer bytes (i.e., without deserialization), the total memory usage of 1.526 GB still exceeds our available budget, amounting to 536% of it. Moreover, deserializing these raw bytes leads to a 931% increase in memory consumption. In contrast, our second approach (with minimized metadata structures) requires only $\approx 277\,\text{MB}$, which is marginally above the calculated budget of $\approx 240\,\text{MB}$. This minor discrepancy may stem from (1) additional memory required for program execution itself and (2) block-based memory allocations that potentially over-provision memory.

8.5 Smaller Row Group Sizes

Overview. We now investigate the effect of reducing row group sizes in our Parquet-based system. In the initial configuration, we selected a row group size of 19,406

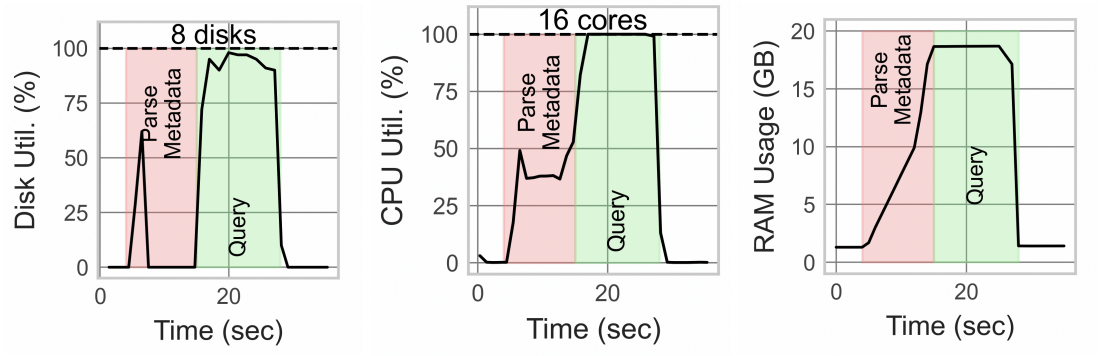

Figure 8.6: Parquet Reader Resource Utilization

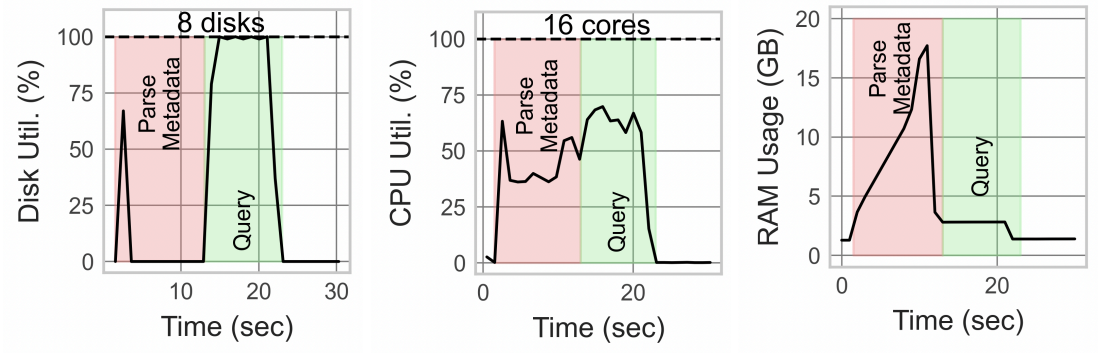

Figure 8.7: Direct Byte–Access Resource Utilization

rows to ensure that all metadata could be stored in memory. However, a natural question arises: What if metadata were only retained in memory for the most frequently accessed files? In such a scenario, smaller row groups could be adopted, thereby enhancing filtering granularity. To explore this idea, we evaluated the performance implications of various factors, such as row group selectivity, row group size, column selectivity, and feature inclusion, for our *Parquet Reader* approach. Our analysis yielded the for key findings: (1) *Metadata Deserialization Time:* Deserializing the metadata imposes a significant time cost. (2) *Selectivity and CPU Bottlenecks:* When selectivity is low: 75% or above, the system becomes CPU-bound rather than disk-bound. (3) *Excessive Metadata Memory Requirements:* The memory footprint for storing all metadata greatly exceeds any reasonable limit. (4) *Substantial Metadata Storage Overhead:* The space required to store metadata on disk is disproportionately large.

Test setup. To illustrate these findings, we designed an experiment in which each approach first reads the files metadata footer, deserializes it (except for the second experiment), and for our *Direct Byte-Access* approach converts it into a minimized metadata structure. We employed a row group size of 550 rows and used a selectivity of

100% (worst-case scenario, where no row groups can be filtered out). Sixteen files were queried simultaneously (one per thread), resulting in a total dataset size of 14,394.88 GB, of which 1,268.48 GB was consumed by raw metadata: approximately 8.81% of the total storage requirements.

Result Discussion. In both approaches, the process of deserializing metadata consistently required around 9 seconds, a duration that is problematic in a distributed storage context and thus forces in-memory metadata storage. During this phase, we observed a brief spike in disk utilization from reading the metadata footer, followed by high CPU utilization due to the deserialization process. Overall memory usage rose significantly, to approximately 16.524 GB, which exceeds the datasets actual size. By contrast, when using the minimized metadata structure in the second approach, memory requirements fell sharply before the query execution phase began. During the data retrieval process, the Parquet reader in the first approach was largely CPU-bound. This behavior stemmed from the need to decompress and deserialize a large number of small row groups, causing the system to operate at 100% CPU utilization while the disk remained underutilized. These findings underscore the challenges associated with small row group sizes and highlight an important trade-off: Larger Row Groups: (1) Reduced storage and memory overhead (2) Faster metadata deserialization (3) Reduced CPU overhead due to parsing fewer row groups (4) Lower filtering granularity Smaller Row Groups: (1) Increased filtering granularity (2) Higher storage and memory overhead (3) Slower metadata deserialization (4) Greater CPU overhead arising from handling numerous small row groups

8.6 Summary and Conclusion

Our evaluation has shown that near-storage query processing on Parquet files can substantially boost efficiency by selectively filtering row groups and columns before reading data. We observed that: (1) *Throughput* improves dramatically (up to 5281% for highly selective queries) when row groups and columns can be skipped; (2) *Different Workloads and Query Selectivities* exhibit exponential increases in throughput as selectivity decreases, meaning highly selective queries benefit most from in-storage filtering; (3) *CPU Utilization* is significantly lower with *Direct Byte-Range Access* owing to minimal metadata parsing and reduced decompression overhead; and (4) *RAM Usage* also benefits under the second approach, as its minimized metadata structures avoid the large memory footprints common to fully deserialized Parquet metadata. When generalizing to a variety of workload conditions, both approaches met or exceeded realistic retrieval rate requirements, as measured against real-world selectivity distributions. At a fixed 10% query selectivity, the *Parquet Reader* approach used more CPU (75–90%) and memory, whereas *Direct Byte-Range Access* stayed between 60–75% CPU usage and required less peak RAM. Nonetheless, there are trade-offs. While *Direct Byte-Range Access* outperforms in CPU overhead and memory usage, it returns partial data (i.e.,

row group contents) rather than a fully intact Parquet file, requiring the client to reconstruct the file if necessary. Furthermore, experiments with *Smaller Row Groups* revealed higher overheads for metadata storage and deserialization, often rendering the system CPU-bound. Although smaller row groups can offer more granular filtering, they incur greater parsing costs and larger metadata files, underscoring that row group size must be selected to balance filtering benefits against these overheads. In conclusion, our findings validate that near-storage computation on Parquet files can achieve significant performance gains and high retrieval rates under real-world conditions. While *Parquet Reader* and *Direct Byte-Range Access* both capitalize on skipping irrelevant data, the latter demonstrates consistently lower CPU and memory footprints, at the cost of only returning partial data rather than a fully reconstructed file.

9 Future Work

While our investigation demonstrates the potential of near-storage query processing for Parquet files, there remain several avenues for further exploration and enhancement:

Evaluating Additional Datasets. Our performance analysis largely focused on representative workloads drawn from Snowset and Thesios. Extending these experiments to a broader range of real-world datasets, especially those with varying schema complexities, skewed distributions, or highly nested data, would provide stronger evidence of the generalizability of our approaches.

Metadata Compression. Although our second approach already employs a compact metadata structure, further reduction in the memory footprint could be achieved through targeted compression techniques. Compressing metadata in at rest could reduce memory overhead and allowing us to further reduce row groups sizes for better granularity. Implementing and measuring the performance of specialized compression algorithms tailored to the minimized metadata structure would be an important next step.

Supporting Additional File Formats. Our system currently supports Parquet as the primary columnar format. Investigating how our near-storage techniques would transfer to other widely used data formats such as ORC, PAX could validate the approach. Different storage layouts and metadata schemas may require adaptations of the filtering, deserialization, and indexing strategies presented here.

Exploring Durability Strategies. Though our experiments assumed that data and metadata are stored reliably in the underlying object store, real-world scenarios often demand robust durability and fault-tolerance mechanisms. Future work could examine how replication, erasure coding and other techniques would fit into our proposed architecture. Balancing high-speed query execution against the overhead of maintaining strong consistency guarantees remains a key challenge in decentralized systems. Taken together, these directions outline a broader research scope aimed at further improving scalability, reducing resource usage, and generalizing near-storage query execution techniques to diverse file formats and operational settings.

10 Conclusion

We analysed that a distributed storage system built on d3en EC2 instances can offer storage costs comparable to S3. We outlined a feasible architecture and estimated its theoretical throughput using benchmarks on d3en instances. By examining the Snowset dataset from Snowflake and the Thesios dataset from Google, we found that a distributed warehouse solution must support an average monthly retrieval rate of 4.842 and 17.84 times, respectively, with query selectivity distributions predominantly near 0%. Furthermore, we introduced two techniques to prune Parquet files during read operations, thereby reducing I/O overhead and boosting throughput. Our results confirm that these approaches achieve a retrieval rate well above the derived real-world requirements.

Bibliography

[1] B. Dageville et al. "The Snowflake Elastic Data Warehouse." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2016, pp. 215–226.

[2] A. Behm et al. "Photon: A Fast Query Engine for Lakehouse Systems." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2022, pp. 2326–2339.

[3] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. "Amazon Redshift and the Case for Simpler Data Warehouses." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1917–1923.

[4] J. Aguilar-Saborit and R. Ramakrishnan. "POLARIS: The Distributed SQL Engine in Azure Synapse." In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3204–3216.

[5] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. "Dremel: Interactive Analysis of Web-Scale Datasets." In: *Proceedings of the VLDB Endowment (PVLDB)* 3.1 (2010), pp. 330–339.

[6] S. Melnik et al. "Dremel: A Decade of Interactive SQL Analysis at Web Scale." In: *Proceedings of the VLDB Endowment (PVLDB)* 13.12 (2020), pp. 3461–3472.

[7] N. Armenatzoglou et al. "Amazon Redshift Re-invented." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2022, pp. 2205–2217.

[8] M. Kuschewski, D. Sauerwein, A. Alhomssi, and V. Leis. "Btrblocks: Efficient columnar compression for data lakes." In: *Proceedings of the ACM on Management of Data* 1.2 (2023), pp. 1–26.

[9] C. Ji, Y. Li, W. Qiu, U. Awada, and K. Li. "Big data processing in cloud computing environments." In: *2012 12th international symposium on pervasive systems, algorithms and networks*. IEEE. 2012, pp. 17–23.

[10] X. Yu, M. Youill, M. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker. "PushdownDB: Accelerating a DBMS using S3 computation." In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, pp. 1802–1805.

[11] Y. Li, J. Lu, and B. Chandramouli. "Selection Pushdown in Column Stores using Bit Manipulation Instructions." In: *Proceedings of the ACM on Management of Data* 1.2 (2023), pp. 1–26.

[12] Amazon Web Services. *Amazon S3 Pricing*. Accessed: 22 Feb. 2025.

[13] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig, et al. "Amazon Redshift re-invented." In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 2205–2217.

[14] I. Pandis. "The evolution of Amazon redshift." In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 3162–3174.

[15] B. Dageville et al. "The Snowflake Elastic Data Warehouse." In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 215226. ISBN: 9781450335317.

[16] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia. "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics." In: *Proceedings of CIDR*. Vol. 8. 2021, p. 28.

[17] Snowflake. *Snowset Dataset*. Accessed: 19 Jan. 2025.

[18] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. "Building an elastic query engine on disaggregated storage." In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 449–462.

[19] G. Research. *Thesios Dataset Repository*. Accessed: 19 Jan. 2025.

[20] P. M. Phothilimthana, S. Kadekodi, S. Ghodrati, S. Moon, and M. Maas. "Thesios: Synthesizing Accurate Counterfactual I/O Traces from I/O Samples." In: ASPLOS '24. Association for Computing Machinery, 2024. ISBN: 9798400703867.