

Vrije Universiteit Amsterdam

Universiteit van Amsterdam



Master's Thesis

An analysis on making DuckDB more friendly to S3 with differential storage

Author: Duc Anh Phan (2836912)

1st supervisor: Peter Boncz
daily supervisor: Boaz Leskes (MotherDuck)
2nd reader: Daniele Bonetta

*A thesis submitted in fulfillment of the requirements for
the joint UvA-VU Master of Science degree in Computer Science*

August 13, 2026

Abstract

DuckDB is an in-process analytical database designed for fast local disks: it keeps the database in a single mutable file and reads it synchronously in small blocks. Cloud object storage breaks these assumptions, as objects are immutable, each request has tens of milliseconds of latency, and the available bandwidth is achieved only with requests of several megabytes and hundreds of them in flight. Differential storage resolves the immutability problem by representing the database file as immutable layers on cloud object stores. However, when used naively, each read still results in a single small request, keeping throughput below network capacity. This thesis investigates how to close that performance gap and finds that doing so requires changes to both the write and read paths. Because differential storage allows placing blocks anywhere in cloud object storage, the physical layout can be reorganized so that the data a scan reads is co-located, enabling large requests. The read path must then be able to take advantage of the new layout: DuckDB's row-group reordering logic is repurposed both to make optimistically cached data reusable and to prefetch ahead of the scan, coalescing the blocks it will need into fewer, larger requests. Results showed that at a scale factor of 500 on a 100 Gbps instance, the best approach reduces the TPC-H workload from 1147 to 104 seconds and TPC-DS from 1969 to 375 seconds, issues more than 45 times fewer S3 requests, and lowers the cost of a run.

Preface

AI tools (Grammarly ¹ and Claude Opus 5 ²) were used in this thesis. Their uses were limited to proofreading the text and suggesting rephrasings for clarity. The design of the system, the experimental setup, the measurements, and their interpretation are the author's original work. The author remains responsible for verifying and interpreting the content, and all core findings and arguments reflect original academic work.

¹<https://www.grammarly.com/>

²<https://www.anthropic.com/news/claude-opus-5>

Contents

Preface	iii
1 Introduction	1
1.1 Overview	1
1.2 Contributions and research questions	2
1.3 Outline	4
2 Background	5
2.1 Database storage	5
2.1.1 Physical storage	5
2.1.2 Pages and Blocks	7
2.1.3 Block data layout	7
2.1.3.1 N-Ary Storage Model (NSM)	8
2.1.3.2 Decomposition Storage Model (DSM)	9
2.1.3.3 Partition Attributes Across (PAX) and columnar storage	9
2.2 DuckDB	9
2.2.1 Overview	9
2.2.2 DuckDB file format	10
2.2.2.1 Headers and management of data blocks	10
2.2.3 DuckDB table storage format	13
2.2.4 DuckDB table scan	17
2.3 Cloud object storage	18
2.4 Differential storage in MotherDuck	20
2.4.1 Motivation	20
2.4.2 Design: immutable snapshot layers	20
2.4.3 Features enabled by differential storage service	22
2.4.4 Caching	22

CONTENTS

3	Literature review	25
3.1	File formats in the cloud	25
3.1.1	Parquet and ORC	25
3.1.2	Future-proof File Format - F3	27
3.2	Pixels: a columnar store for the cloud	28
3.2.1	PAX or DSM	29
3.2.2	Request size and reading method	30
3.2.3	Discussion	31
3.3	AnyBlob: exploiting cloud object stores for OLAP workloads	31
3.3.1	Characterizing cloud object storage	31
3.3.2	Efficient I/O engine	34
3.3.3	Anyblob and integration	34
3.3.4	Discussion	35
3.4	Putting it all together	35
4	Motivation and design	37
4.1	Revisiting object storage performance	37
4.1.1	Latency	38
4.1.2	Request sizes and throughput	39
4.1.3	Implications for the design	40
4.2	An analysis of differential storage layout on S3 - TPC-H, TPC-DS, and a MotherDuck workload	41
4.2.1	Methodology	41
4.2.2	TPC-H and TPC-DS	41
4.2.3	A real-world workload at MotherDuck	51
4.2.4	Summary	53
4.3	Write path	54
4.3.1	A simple S3 file system	55
4.3.2	Single buffer	57
4.3.3	Per-column buffer	57
4.3.4	Compaction	58
4.4	Read path	58
4.4.1	Synchronous read	59
4.4.1.1	Without caching	59
4.4.1.2	With caching	60

4.4.2	Reorderer	61
4.4.3	Prefetching	62
4.4.3.1	Without batching	64
4.4.3.2	With batching	64
4.5	Memory considerations	65
4.6	Putting it all together	66
5	Performance evaluation	69
5.1	Evaluation setup	69
5.1.1	Benchmark suites	69
5.1.2	Compute and storage infrastructure	70
5.1.3	Notation	70
5.2	Performance criteria	71
5.3	TPC-H	71
5.3.1	Vanilla DuckDB	72
5.3.2	Synchronous read without caching, on the single-buffer and per-column layouts	72
5.3.3	Synchronous read with caching, on the single-buffer layout	74
5.3.4	Synchronous read with caching, on the per-column and compacted layouts	78
5.3.5	Synchronous read with caching and the modulo reorderer	78
5.3.6	Prefetching	79
5.3.6.1	Normal prefetching	79
5.3.6.2	Batch prefetching	80
5.3.6.3	Cost	81
5.3.7	Putting it all together	83
5.4	TPC-DS: does the trend hold?	86
5.4.1	Differential storage versus vanilla S3	86
5.4.2	Synchronous read with caching	87
5.4.3	Prefetching	87
5.4.4	Putting it all together	89
5.5	Summary of the results	91

CONTENTS

6	Discussion	97
6.1	Reflecting on research questions and related work	97
6.1.1	Research questions	97
6.1.2	Related work	99
6.2	Future work	99
7	Conclusion	101
	References	105

1

Introduction

1.1 Overview

In the past decade, cloud-based data warehouses such as Snowflake, Google BigQuery, Amazon Redshift, Databricks, and ClickHouse Cloud [20; 23; 39; 16] have emerged, efficiently serving analytical workloads. The move from on-premises infrastructure to the cloud is motivated by the scalability and elasticity of this environment [23; 41], achieved by decoupling compute and storage. *Cloud object storage*, such as AWS’s S3, Google’s Cloud Storage, and Microsoft’s Azure Blob Storage, is central to this separation. It stores and distributes data across thousands of disks and offers advantages over block storage (such as AWS EBS, Azure Disk Storage [7; 34]) or file systems (such as AWS EFS, Azure Files [8; 35]), including cost-effectiveness, high aggregate throughput, and reliability.

DuckDB is an in-process analytical database designed for fast, interactive SQL queries on local files. A core challenge in integrating DuckDB with the cloud is that DuckDB’s storage format is optimized for local SSDs, not for cloud object storage such as S3. DuckDB uses a 256 KiB block size, which is small for efficient S3 usage, and stores all its data in a single continuously modified file. This file layout is incompatible with object storage design, which does not support in-place modification or efficient handling of small, frequent writes.

To address this, MotherDuck introduces a key infrastructure called *differential storage* [30]. Differential storage acts as an intermediary between DuckDB’s file system interface and S3, translating the local file operations into S3-compatible actions. This indirection layer overcomes object storage immutability constraints and enables features like *snapshots*, *data sharing*, *time travel* and *zero-copy-on-write*, making DuckDB cloud-native.

However, naively using differential storage by issuing small reads to S3 would result in suboptimal query performance, leaving room for further optimization. More specifically,

1. INTRODUCTION

S3 is beneficial for bulk reads and multiple concurrent requests. This gives us ways to optimize the system, namely by reorganizing the data layout, enabling efficient prefetching, sizing row groups, and coalescing S3 requests. In this thesis, we examine current differential storage behaviors and experiment with approaches to improve system performance. We show that a per-column layout, cleaned up by a larger row group and offline compaction, combined with reorderer-based prefetching reduces execution time by roughly $11\times$ on TPC-H and $5\times$ on TPC-DS over synchronous reads, while cutting S3 request counts by more than $45\times$ on both.

1.2 Contributions and research questions

In this thesis, we aim to answer three research questions.

RQ1: What is the current access pattern of DuckDB with differential storage, and what is the S3 throughput we can currently achieve?

In this research question, we first examine DuckDB’s access patterns on local storage. We use TPC-H and TPC-DS, standard benchmarks for database systems, to analyze the workload patterns. We want to understand how DuckDB writes different types of data, including metadata and actual column data. More specifically, we ask what kind of data is written within a data block, e.g., whether it belongs to the same row group, the same column, or different tables. From this, we study the access pattern on object storage. Because differential storage introduces a layer of indirection between DuckDB and S3, there are now two perspectives on file operations: the logical reads and writes as seen by DuckDB, and the physical reads and writes as performed on S3. In our analysis, we distinguish between these two types of operations. We focus on how aggregating logical reads into larger physical S3 requests depends on the layout and locality of data within S3 files. For example, if related data (such as from the same table, row group, or column) is stored contiguously, it becomes possible to coalesce multiple logical requests into fewer, larger S3 operations, improving efficiency. We study this using both the standardized benchmarks TPC-H and TPC-DS and a real-world workload at MotherDuck. Furthermore, we benchmark the throughput achievable when reading under the current scheme, compared to the line-rate throughput that cloud services can offer.

RQ2: What changes could one make to differential storage’s write path to improve S3 throughput?

After studying access patterns, this question aims to determine how to organize data on S3 to reduce fetch time and costs. The design of differential storage offers a unique

1.2 Contributions and research questions

opportunity to reorder data without changing DuckDB’s file format. DuckDB requests the differential storage system to write data in 256 KiB blocks, which means that differential storage can decide where to put these blocks on cloud object stores. The naive scheme appends blocks to a single buffer shared across threads. Once the buffer accumulates 16 MiB or 64 blocks, it is persisted as a single blob to S3. However, if we incorporate more context into a write, i.e., if we let differential storage know what kind of data this is (metadata, data from which column of which table), it could, for example, arrange blocks from the same column together (per-column layout). A key obstacle here is *multi-segment blocks*, i.e., blocks that pack data from multiple columns, which we quantify and mitigate by using larger row groups. We can also perform offline compaction to get the best layout. All of these allow differential storage to read a 16 MiB S3 file instead of just one block at a time. We discuss the necessary hooks into DuckDB and evaluate the performance. We then study the effect of this layout on DuckDB’s current synchronous read path.

RQ3: What changes could one make in DuckDB’s read path to improve S3 throughput, and how do they interact with the layout from RQ2?

DuckDB currently does not support asynchronous I/O. The number of threads available bounds the number of concurrent I/O requests. Cloud providers, especially AWS, offer a wide range of network instances with high throughput from 25 Gbps to 100 Gbps, or even hundreds of Gbps for extremely high-end, accelerated instances. To saturate the network bandwidth of these instances, we need to issue hundreds of S3 requests at higher granularities such as 8 MiB or 16 MiB. Therefore, we explore the possibility of prefetching via a special hook called *reorderer* in DuckDB, which normally reorders row-group scans but which we repurpose to fetch row groups ahead of time. We evaluate this both with and without the per-column layout from RQ2. We expect prefetching to remove the concurrency bottleneck, but improving request size still requires the per-column layout of RQ2.

We also survey efforts to bring analytical databases onto cloud object stores. Overall, the five main contributions of this thesis are:

1. An empirical characterization of the storage layout across TPC-H, TPC-DS, and a real workload at MotherDuck
2. A per-column write path design and analysis on compaction opportunities
3. A caching strategy that maximizes the cache hit rate
4. Reorderer-based prefetching with request coalescing

1. INTRODUCTION

5. A multi-metric evaluation across approaches, covering execution time, cost, S3 request count and size, bytes read, and cache hit rate

1.3 Outline

In Chapter 2, we discuss essential topics to understand the rest of this thesis. We first walk through common definitions of databases and various storage models. We then go deeper into how DuckDB organizes its data and how it interacts with the file system interface. Next, we characterize cloud object storage services and the reasons they are a reasonable choice for storing data. Finally, we discuss the architecture of differential storage.

In Chapter 3, we present related work that aims to store data on blob storage services. We first go through some standard file formats that are used in cloud-based data warehouses. We move to discuss two papers (AnyBlob [27] and Pixels [18]) that aim to improve the performance of query execution on S3-backed storage systems. Finally, we discuss which directions to go to improve our differential storage system.

In Chapter 4, we present our main work separated into five parts. First, we present our analysis of differential storage behavior on standard benchmarks and a real-world workload at MotherDuck to motivate our approaches. We then show how we would reorganize data layout on S3 via the write path. Next, we present our first attempt to improve performance using synchronous reading with optimistic read-into-cache. We further improve the system by achieving asynchronous reading via prefetching using the reorderer logic within DuckDB. Finally, we discuss how we would implement these ideas into DuckDB.

In Chapter 5, we present the evaluation of our approaches and their implications.

In Chapter 6, we reflect on different approaches and how they answer the research questions and propose future work for this thesis. Finally, we conclude the thesis in Chapter 7.

2

Background

In this chapter, we discuss topics needed to understand various aspects of the thesis.

2.1 Database storage

A traditional computer stores data at different levels, each with distinct characteristics. In this section, we discuss different storage classes within a computer system and their use cases.

2.1.1 Physical storage

Several classes of storage exist within a computer system. This is usually referred to as a storage hierarchy [29], in which storage devices are classified by access speed, as shown in Figure 2.1. Devices closer to the CPU offer faster access times but at a higher cost per byte and lower capacity. A fundamental separation between storage devices is volatility. Non-volatile devices differ from their counterparts in various aspects:

- **Speed:** Speed is broken down into *latency* — the minimum time to access a single byte — and *bandwidth* — the rate at which the CPU can sustain reads from the storage device. Both degrade as devices move away from the CPU, and non-volatile devices are orders of magnitude worse in both cases [29]. A DRAM access completes in tens of nanoseconds. In contrast, a modern NVMe SSD exhibits latencies in the order of tens to hundreds of microseconds, and magnetic disks in the order of milliseconds. Bandwidth decreases similarly: a single DRAM subsystem achieves tens to hundreds of GB/s, while a modern NVMe SSD reaches around 7 GB/s per device [28] and a magnetic disk only delivers hundreds of MB/s [29].

2. BACKGROUND

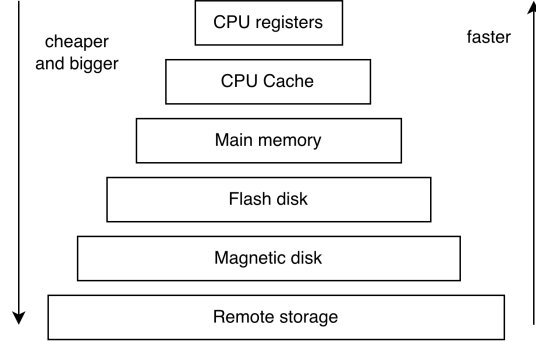


Figure 2.1: The storage hierarchy. Capacity grows and cost per byte drops at each step down, but latency increases by orders of magnitude. That is why where the data stays matters.

- **Durability:** Non-volatile devices retain their data across crashes, whereas the contents of CPU caches and DRAM are lost when power is removed [29]. Durable data must therefore ultimately reside on non-volatile storage; volatile memory can only act as a cache over it.
- **Access granularity:** Volatile storage is byte-addressable and accessed directly by the CPU through load or store instructions. Non-volatile devices instead expose a block interface, where data is read and written in fixed-size blocks (typically 4 KiB). I/O requests are issued via operating system calls. Modifying a single byte would then require reading, updating, and rewriting an entire block.

Non-volatile storage typically favors sequential access over random access. For magnetic hard drives, this is because of their design: every random access requires seeks and rotations, making random I/O orders of magnitude slower than sequential scans. Flash-based SSDs have no moving parts and exploit internal parallelism, so the performance difference between sequential and random access is considerably smaller. Random reads on modern NVMe devices approach sequential bandwidth [28].

When moving to a cloud environment, database systems prefer to store data on an object storage service. Cloud object storage has unique characteristics, namely, much higher latency, support for higher-granularity requests, unlimited capacity, and high throughput [27]. S3 service throughput can exceed that of a single NVMe. Accessing data via object storage therefore requires a different approach than reading from local disks. We will characterize S3 services in the subsequent section.

2.1.2 Pages and Blocks

Data can reside at different layers of the storage hierarchy, but users generally expect their data to persist. Consequently, when designing database systems, one needs to consider how data is organized on and accessed from persistent storage. In-memory databases have their own use cases but are out of scope for this thesis. Moving data between disk and memory is a core task of every disk-based database system. Since accessing a disk block is orders of magnitude slower than accessing memory, systems should be designed to avoid I/O stalls. Data is commonly split into fixed-size blocks called pages, stored in one or more files. A DBMS usually does not rely on operating systems via, for example, `mmap`, to manage the transfer of data between disks and memory, but rather has its own buffer management logic (i.e., buffer pool) [22]. Each page can contain different types of data, namely metadata, indexes, or the data itself, and is usually assigned a unique ID. A database has its own structure for managing the allocation and use of disk pages. If a DBMS stores data in a single file, such as DuckDB [26], the page ID is usually just the offset in the file. Meanwhile, for data stored across multiple files, such as in PostgreSQL [40], databases need to manage page IDs themselves. For example, they add another layer of indirection to translate logical page IDs into physical positions in the file system.

Calling data blocks “pages” should not be mistaken for pages in operating systems. In this thesis, we use page and block interchangeably, but when discussing DuckDB we primarily refer to them as blocks. Pages in operating systems are typically 4 KiB, while pages in DBMSs can vary depending on the design. Database systems with read-intensive workloads tend to have larger page sizes (256 KiB–1 MiB), while write-intensive database systems have smaller pages (4 KiB–16 KiB) [37].

It is important to note that not all systems organize data into fixed-size pages. Umbra employs variable-size pages [36], and database systems using log-structured storage such as RocksDB organize data into variable-size units [25]. DuckDB uses fixed-size pages to manage storage space, so in this thesis, we focus only on fixed-page-based database systems.

2.1.3 Block data layout

Pages or data blocks are the unit DBMSs use to read and write from disks. The next step is to decide how we organize rows into a page. Different workloads would require different schemes to arrange data tuples within a page. Workloads for databases are categorized into three types:

2. BACKGROUND

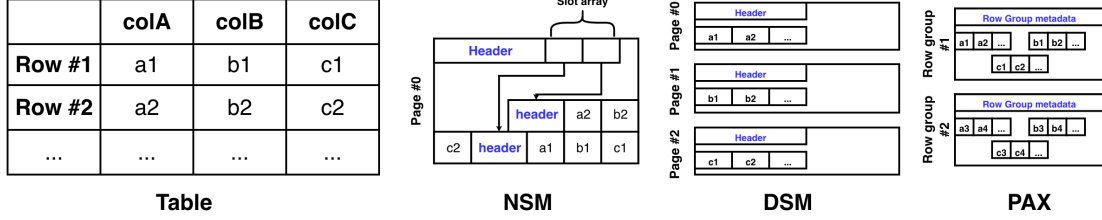


Figure 2.2: Illustration of different storage models. (1) NSM keeps tuples contiguous and suits OLTP. (2) DSM stores each column separately and suits OLAP. (3) PAX: a fixed number of rows is stored per page, with the values within a page grouped column by column.

- **Online Transaction Processing (OLTP):** OLTP workloads are characterized by frequent writes and by transactions that each touch only a small amount of data.
- **Online Analytical Processing (OLAP):** OLAP workloads are read-heavy and scan-oriented: each query typically reads large portions of the data but writes little, often through bulk appends rather than in-place updates.
- **Hybrid Transaction and Analytical Processing (HTAP):** HTAP workloads combine both patterns, serving transactional writes and analytical scans over the same data, so the system must handle small point operations and large scans at the same time.

We now discuss how different page layouts benefit the workloads described above. Figure 2.2 illustrates how different data models organize data on pages.

2.1.3.1 N-Ary Storage Model (NSM)

The N-Ary Storage Model stores all the attributes of a row consecutively within a page, one record after another. Therefore, it is also called a row store. OLTP workloads favor NSM, as they involve accessing and updating a single record. It only takes one fetch to get all the fields from a record. However, this comes with downsides in OLAP workloads, in which scanning only a few attributes results in a full read of all attributes, leading to inefficient I/O and poor cache performance [1]. Compression also becomes harder because of multiple data types within a data block.

2.1.3.2 Decomposition Storage Model (DSM)

DSM is purely columnar, with columns stored on their own pages [21]. DBMSs construct a row from different pages when needed. DSM exhibits high I/O throughput and good cache performance [1]. The cost is single-record access: accessing all attributes of a row requires reading multiple blocks for all columns, whereas with NSM it requires reading only one block. This is why it is not ideal for point queries, inserts, or updates.

2.1.3.3 Partition Attributes Across (PAX) and columnar storage

PAX [2] is a hybrid scheme that vertically partitions attributes within a page. In PAX, data is horizontally partitioned into groups of rows. Within each row group, the attributes are vertically partitioned into columns, so that each row group behaves like a small column store over its subset of the rows. PAX is widely adopted today in variants such as Parquet and ORC. These are usually referred to as columnar storage formats [44]. PAX-inspired storage keeps the benefits of a pure columnar store within a row group, such as cache locality and efficient I/O, since a row group's columns are co-located and positionally aligned. A further advantage is that row groups are self-contained and self-describing, allowing the DBMS to manage them independently without complex coordination.

This model is central to the rest of this thesis: DuckDB organizes its data as row groups of fixed row count, with each column stored separately within a row group as one or more pages, making its on-disk format a PAX variant [26].

2.2 DuckDB

2.2.1 Overview

DuckDB is a database system created at CWI [38], primarily serving analytical workloads. It has attracted significant attention within the DBMS category for its high speed, rich SQL dialect, broad community support, and extensibility. DuckDB is intended to run on a local machine with fast local storage, an assumption that shapes its I/O model and that this thesis revisits in the context of remote object storage.

Recalling the previous sections, we can characterize DuckDB as follows:

- **DuckDB is an OLAP database system.**
- **Page-based data layout:** data is split into blocks of 256 KiB, and DuckDB interacts with the file system at this granularity. This aligns with most of the disk-based

2. BACKGROUND

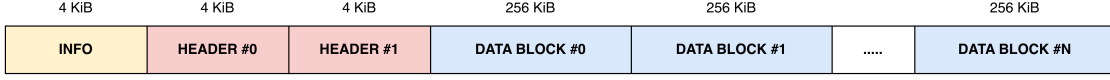


Figure 2.3: Block layout in DuckDB file. The two-header design enables single-file storage with atomicity, in which the database state is switched via a single header write. Only one header is active at a time, representing the state of the database.

database systems we discuss in Section 2.1.2. From now on, we refer to pages in DuckDB as data blocks, or simply blocks.

- **Columnar (PAX-style) storage:** data is split into row groups of fixed row count, and data within a row group is stored columnarly.

2.2.2 DuckDB file format

We now discuss in detail how DuckDB implements the combination of fixed-size page layout and columnar storage.

DuckDB uses a single-file, page-based format. This means that data is stored within a single file, which is further split into fixed-size blocks. The physical layout of a DuckDB file is simple, as shown in Figure 2.3. There are three 4 KiB blocks located at the beginning of the file:

- **Info block (offset 0 to 4 KiB):** this metadata block contains information about the version, including DuckDB version, DuckDB storage version, and other information such as encryption metadata.
- **The two header blocks (offset from 4 KiB to 8 KiB and 8 KiB to 12 KiB):** these two blocks are the actual headers and contain pointers to other metadata of the database, such as pointers to catalog entries or pointers to the free list, which is used to manage free blocks.
- **Data blocks:** regular blocks that store both **metadata** and **data**, 256 KiB in size, starting at offset 12 KiB. The block ID is an index that maps to the offset within the file, starting at 0 (offset = 12 KiB + ID × 256 KiB). From now on, we refer to them as **data blocks** or **blocks** for short, not to be confused with **meta blocks**, which are only 4088 bytes and reside in a data block (see Figure 2.4).

2.2.2.1 Headers and management of data blocks

This section discusses how pages or data blocks in a single file are managed in DuckDB.

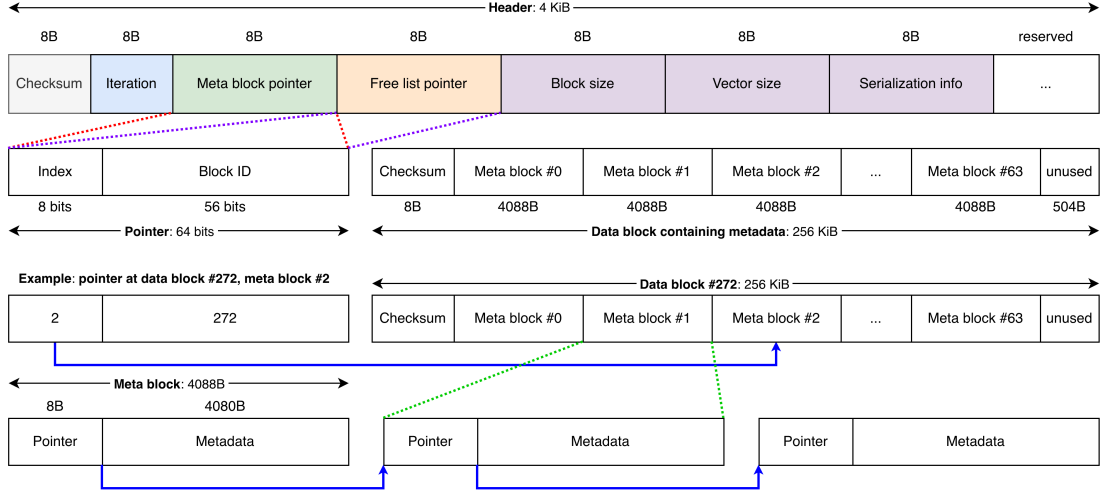


Figure 2.4: Header and meta block breakdown. The figure explains how metadata is stored by chaining multiple 4088 B meta blocks within 256 KiB blocks, starting with a pointer. The red-dotted and purple-dotted lines expand the meta block pointer and free list pointer fields, respectively, showing that a pointer is a 64-bit value packing a 56-bit block ID with an 8-bit index into the meta blocks. A 256 KiB data block holding metadata begins with an 8 B checksum and is then divided into 64 meta blocks of 4088 B each, leaving 504 B unused. The example illustrates a pointer to meta block #2 within data block #272. The bottom layer shows how meta blocks are chained to store metadata larger than 4080 B: each meta block reserves its first 8 B for a pointer to the next meta block, leaving 4080 B for payload, and the chain may continue into meta blocks residing in other data blocks.

Meta blocks. Metadata is not stored directly in 256 KiB data blocks, but in smaller 4088-byte units called *meta blocks*. A 256 KiB data block that holds metadata is subdivided into 64 meta blocks, as illustrated in Figure 2.4. Since metadata usually does not fit into a single meta block, each meta block begins with an 8-byte pointer to the next meta block, forming a linked chain that is terminated by a meta block whose pointer is `-1` (`INVALID_BLOCK`). By traversing this chain, we can obtain all the needed metadata. A pointer to a meta block, such as the meta block pointer and free list pointer in the header, is essentially a data block ID and an index into that data block.

Double-header design. Each database header holds an iteration counter, a pointer to the root metadata block from which the entire database state (catalog, tables) is reachable, and block allocation info (free list). At any moment, exactly one header, the one with the higher iteration number, defines the current, valid state of the file. Checkpointing works as follows: new and modified data is written copy-on-write into free blocks, never overwriting

2. BACKGROUND

anything reachable from the active header. Once all data and the WAL checkpoint marker are safely synced to disk, DuckDB writes the inactive header with an incremented iteration counter pointing at the new root, and syncs it. This write is atomic, and the file transitions from an entirely old state to an entirely new state in a single indivisible step, with no observable in-between. The two-header scheme is what enables ACID at the storage layer: atomicity, because visibility of a checkpoint relies on a single block write; consistency, because no reader can ever observe a mix of old and new metadata; durability, because the `fsync` ordering guarantees that the header never points to unflushed data.

Data operations and checkpointing. DuckDB maintains table data in memory in the same form as on disk. All committed changes are held in various in-memory structures, with durability guaranteed via the WAL. Checkpointing is responsible for persisting these changes to disk (i.e., writing new row groups and truncating the WAL), and it interacts with concurrent transactions in different ways to maintain ACID properties. There are two types of checkpoints: `FULL_CHECKPOINT` and `CONCURRENT_CHECKPOINT`. Full checkpoints include tasks that change the layout of old row groups (i.e., shifting row IDs); hence, they need exclusive access to the current state of the data: no writers are allowed during checkpointing, only read-only transactions. Read-only transactions can safely access the old row-group list even when a concurrent checkpoint switches the in-memory data to a new row-group list: the reader keeps a reference to the old list, and copy-on-write block management (via `shared_ptr`) guarantees that old blocks are not freed for reuse until their reference count drops to zero. Concurrent checkpoints allow concurrent deletes and inserts, since they do not change the underlying row IDs and preserve insert and delete information into the new row-group list (i.e., concurrent delete and insert information) even when it resides in the old state, and is carried over to the new row-group list via proper sharing and locking.

Life cycle of data blocks. Data blocks are managed under a global structure called `SingleFileBlockManager`¹. `SingleFileBlockManager` assigns identifiers to blocks and keeps track of which blocks are in use and which are available. Rather than tracking only a simple “free or used” flag, it maintains several distinct sets that reflect the fact that a block’s reusability depends on the checkpoint lifecycle:

¹https://github.com/duckdb/duckdb/blob/main/src/include/duckdb/storage/single_file_block_manager.hpp

- `SingleFileBlockManager::free_list`: Blocks that are free and can be used immediately reside in the free list;
- `SingleFileBlockManager::newly_used_blocks`: Blocks that were allocated within the current checkpoint but never made durable can be recycled at once
- `SingleFileBlockManager::modified_blocks`: Free blocks that still belong to the last durable version of the database are tracked until the next checkpoint completes, so that a consistent on-disk image always survives a crash.
- `SingleFileBlockManager::multi_use_blocks`: A single 256 KiB block can host more than one column segment, because the `PartialBlockManager` packs several small compressed segments into the same block. Such shared blocks are tracked here as a map from block identifiers to reference counts — the number of segments occupying the block — and the block is returned to the free list only after its last referencing segment is freed, with the exception of the case below.
- `SingleFileBlockManager::free_blocks_in_use`: The reference count above tracks on-disk sharing. A block can be referenced from memory: even if a block is not in `multi_use_blocks`, it can still be referenced by a live thread — for example, an ongoing scan that has loaded the block. When such a block is freed, its identifier must not be considered free while another process is still referencing it, so it is placed here and moved to the free list only after the last thread releases it.

When no free block is available, the manager extends the file by advancing the high-water mark (`max_block`); conversely, it can shrink the file by lowering that mark and trimming the tail once trailing blocks become free. In this way, the `SingleFileBlockManager` acts as the allocator that manages the identities of data blocks. In contrast, the buffer pool independently manages whether any given block's contents are currently resident in memory.

2.2.3 DuckDB table storage format

In this section, we discuss how DuckDB implements the PAX-style format in data blocks. DuckDB has a default block size of 256 KiB as illustrated in Figure 2.3.

2. BACKGROUND

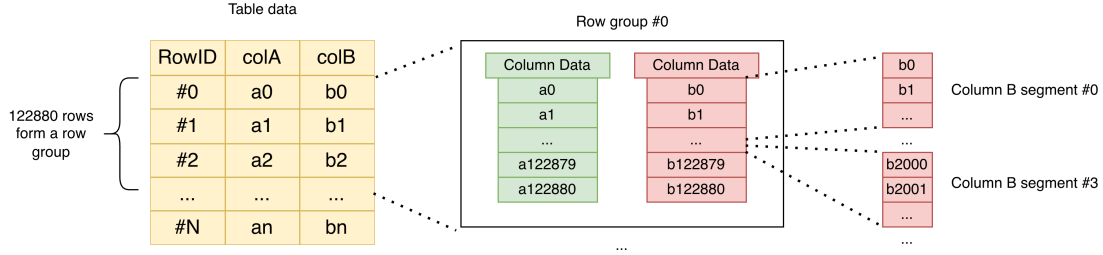


Figure 2.5: Row groups in the DuckDB file format. A table is split into row groups of 122,880 rows, and within a row group each column is stored separately. The row group is the unit of both scanning and checkpointing.

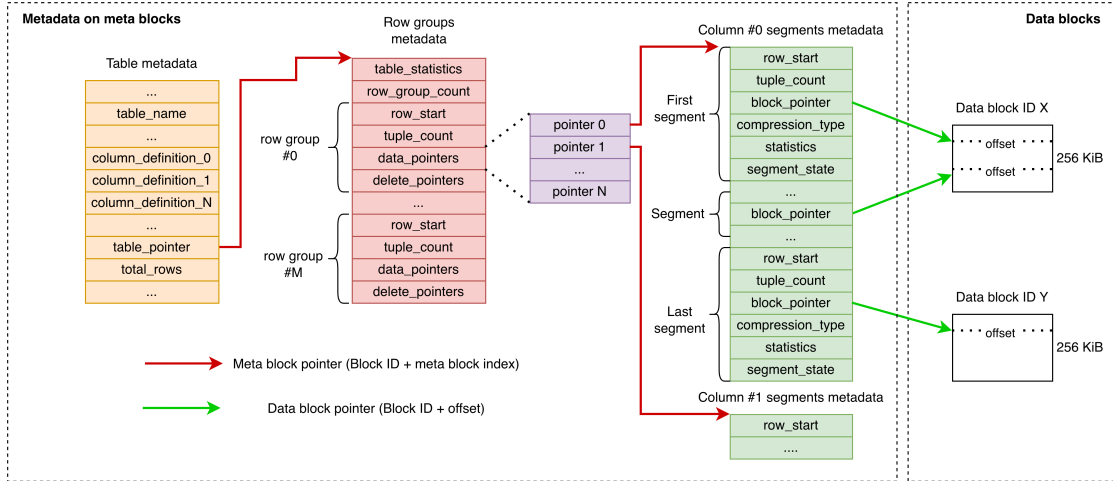


Figure 2.6: DuckDB file format. Metadata is stored in meta blocks, which contain pointers to the actual data held in 256 KiB blocks. A block may hold segments belonging to multiple columns.

Logical view – how data is split. At a logical level, DuckDB organizes data in row groups with the default size of 122,880 rows. Within each row group, each column’s data is further split into segments. Segments are not, per se, at the logical level; we treat them as logical to make them easier to discuss at the physical level. Segments are there to fit data into 256 KiB blocks. This is shown in Figure 2.5.

Physical view – how data is really stored on disk, i.e., the file format. On disk, all metadata and table data stay within 256 KiB blocks. As noted in the previous section, metadata also resides on these 256 KiB blocks, specifically the 4088 B meta blocks. If a piece of metadata does not fit into a single 4088B meta block, DuckDB chains them

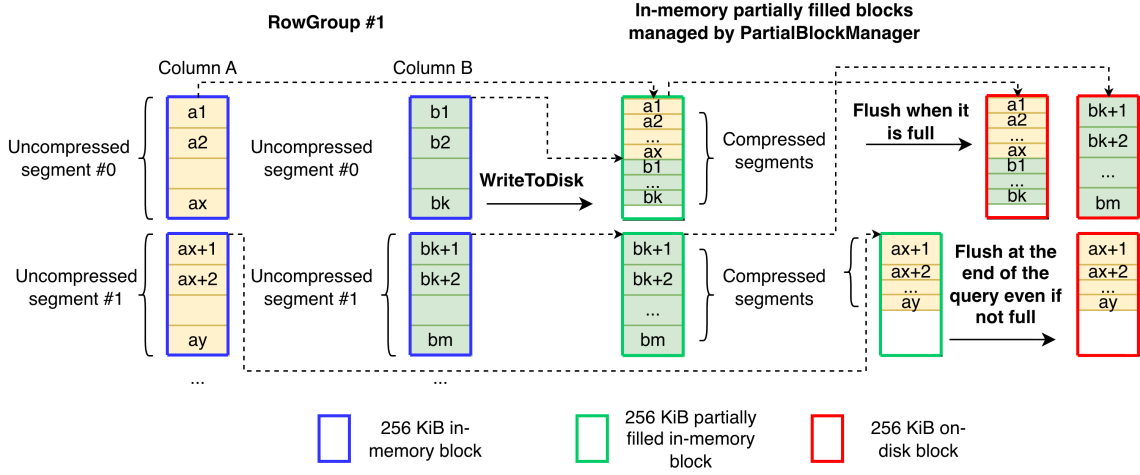


Figure 2.7: High-level overview of data ingestion in the DuckDB native file format. Data blocks are written optimistically, before the transaction commits, in a bulk insert. As a result, the write-ahead log records only lightweight metadata rather than every row.

together to create enough space using a meta block pointer (block ID and meta block offset). Reading a metadata structure might involve traversing this chain of meta blocks. Figure 2.6 illustrates the structure on disk of a table, a rectangle indicating that this data comes from a chain of meta blocks. We start at the catalog information for the table, which stays on meta blocks. This table catalog entry includes information such as the table name, number of columns, column data types, indexes, and, more importantly, a pointer to the row group list. The row group list starts with statistics for all row groups and the total number of rows in this table, followed by a list of structures, each describing a row group. This structure contains the row start ID, number of rows, and a list of meta block pointers. Each pointer represents the location of the metadata for a column of that row group. As column data from a row group is further divided into column segments, this metadata contains a list of structures describing the segments, one after another. Metadata for a segment includes the row start ID, the number of rows, compression information, pruning statistics, and the important block pointer (block ID and offset into the block) that points to the actual data for this segment. In DuckDB, a segment does not necessarily occupy the full 256 KiB block; this depends on how the column data in a row group is compressed. After compression, data can shrink to a point that it only partially occupies a block. DuckDB efficiently addresses this by managing partially-filled blocks and only flushing them to disk in two cases: (1) when the data block is almost full; (2) at the end of the insert operation, when blocks are persisted in the presence of optimistic writes.

2. BACKGROUND

Data ingest and how blocks are persisted. DuckDB has multiple operators to ingest data; the two main ones are `PhysicalBatchInsert`² and `PhysicalInsert`³. The behavior is decided by the size of the insert, which splits ingestion into two paths. A **small insert** — one that ingests less than a full row group — is handled in the simplest way: its data is written directly to the write-ahead log (WAL). The rows reside in memory (uncompressed) and are made durable to the WAL until a later checkpoint puts them to disk. A **bulk insert** — one that has one or more full row groups — cannot afford this: logging every row would make the WAL grow unbounded. Both operators therefore employ *optimistic writing*, in which the data is compressed and written straight into the database file (data blocks) during ingestion, before the transaction commits, while only lightweight row-group references (metadata) are written to the WAL. This keeps the WAL small no matter how much data is bulk-loaded. Figure 2.7 illustrates this. The mechanism works as follows. Data first stays in in-memory 256 KiB blocks, uncompressed. As full row groups form, DuckDB compresses a batch of them together, column by column, by re-iterating over every `DataChunk`⁴ of each column and giving it to the compressor. The compressed output is placed into blocks handed out by a `PartialBlockManager`, which returns either a partially filled block with sufficient free space or a new block, depending on the segment’s compressed size. These blocks are then persisted optimistically in two situations: (1) a block that becomes almost full (at least 80% utilized) is flushed immediately; (2) at the end of the insert, any row groups not yet flushed — complete or under-filled — are written, together with the remaining partially-filled blocks. By default, DuckDB maintains a *per-column* `PartialBlockManager` and waits for a batch of row groups before compressing them all at once, column by column, so that data from the same column is placed into the same persisted block. Any partially-filled blocks left over are then merged into a single global `PartialBlockManager` in `LocalStorage`⁵ and flushed to disk. This also explains why, within a row group’s column data, each segment points to a data block together with an offset into that block, a direct consequence of partial-block management. Further implementation-specific details exist but are out of scope for this background section.

²https://github.com/duckdb/duckdb/blob/main/src/include/duckdb/execution/operator/persistent/physical_batch_insert.hpp

³https://github.com/duckdb/duckdb/blob/main/src/include/duckdb/execution/operator/persistent/physical_insert.hpp

⁴https://github.com/duckdb/duckdb/blob/main/src/include/duckdb/common/types/data_chunk.hpp

⁵https://github.com/duckdb/duckdb/blob/main/src/include/duckdb/transaction/local_storage.hpp

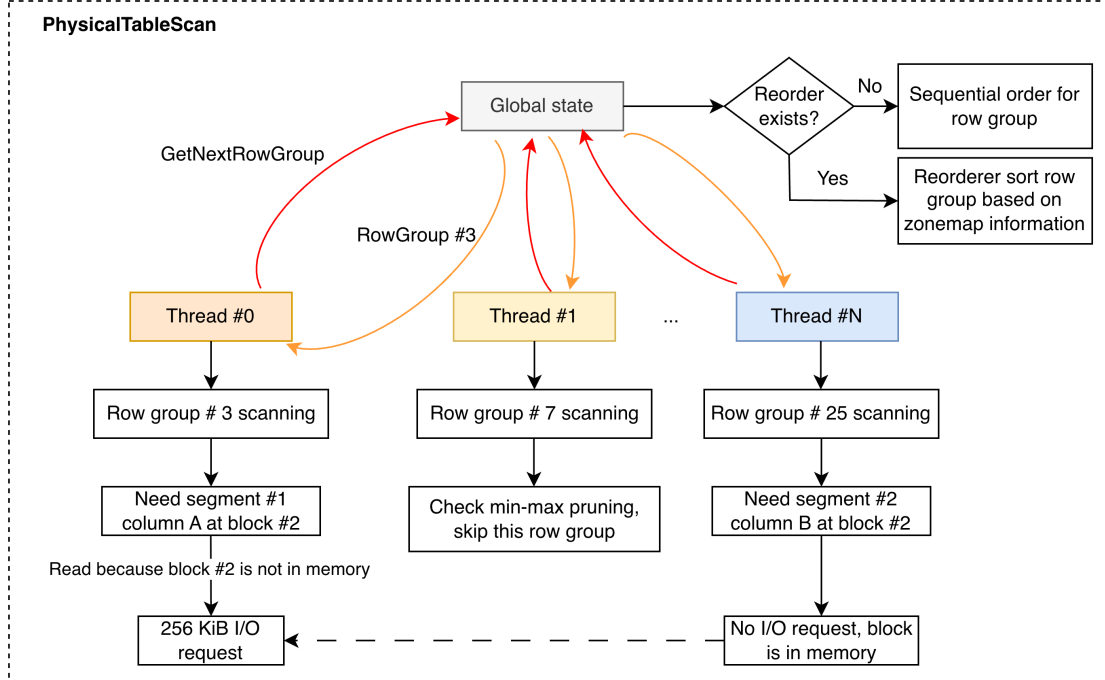


Figure 2.8: DuckDB scan simplified. Worker threads acquire a row group from a global state under a lock. Normally, row groups are handed out in the order in which they appear in the metadata; the exception is the *reorderer*, which sorts the row groups based on additional information and hands them out in that order.

2.2.4 DuckDB table scan

Overview. Figure 2.8 illustrates a simplified view of a DuckDB table scan (see `PhysicalTableScan`⁶ for details). The row group is DuckDB’s unit of parallelism: worker threads repeatedly obtain the next row group from a shared global scan state and process it independently. For each assigned row group, the worker first checks the zonemap (min–max) statistics against the query predicates and skips the row group if no rows can qualify.

Synchronous reads. For native DuckDB tables, all reads are synchronous: when a required data block is not resident in the buffer pool, the worker blocks until the block has been fetched into memory. The number of outstanding I/O requests is therefore bounded by the number of worker threads, far below the hundreds of concurrent requests needed to utilize object storage effectively.

⁶https://github.com/duckdb/duckdb/blob/main/src/execution/operator/scan/physical_table_scan.cpp

2. BACKGROUND

Row-group reordering. A built-in optimizer may install a reordering hook, implemented in `RowGroupPruner` ⁷, into the global scan state to change the order in which row groups are handed to workers. By default, DuckDB dispatches row groups sequentially, following the row group list in the table metadata. For queries with `ORDER BY` and `LIMIT`, however, the optimizer sorts row groups by the min (or max) statistics of the ordering columns, increasing the chance that the scan can terminate early once the `LIMIT` is satisfied. This hook is the single point in the scan pipeline where the dispatch order of row groups can be controlled, which makes it a natural place to schedule row groups for prefetching; we exploit this in Chapter 4. DuckDB currently does not support plugging in custom reorderers.

2.3 Cloud object storage

Cloud object stores (blob storage), such as AWS S3, Azure Blob Storage, and GCP Cloud Storage, are the standard distributed storage backends of cloud database systems: cloud-native data warehouses such as Snowflake [23], Amazon Redshift [16], and Databricks [17] all use object storage as their data source. Object stores offer virtually unlimited capacity with strong durability guarantees at a low storage price of roughly \$23/TiB per month [9], making them both the most durable and the cheapest storage option in the cloud [27].

How does cloud object storage work? There is little information that discloses the internal design of object storage services. However, AWS, the industry-leading cloud provider, has revealed some aspects of the design of its S3 system. In a talk at AWS re:Invent 2024 [33], AWS noted that the physical storage devices used to store data are hard disk drives (HDDs). HDDs are cheap yet slower than solid-state drives (SSDs). Thus, to deliver comparable throughput when reading, objects themselves in S3 are sharded and distributed across a large number of diverse disks, even if a customer’s storage usage requires only a few disks. AWS refers to this as shuffle sharding, and they can do this cheaply because their disk fleet is huge and shared among customers. The problem of placing data chunks from objects onto disks is also interesting, as a strategy can directly affect performance. For example, decorrelating data segments across customers can help avoid a burst of workload when two customers request simultaneously. An important result of sharding is that object storage can saturate the network at a high rate, as fetching multiple distinct objects is fully parallelizable as they reside on different disks.

⁷https://github.com/duckdb/duckdb/blob/main/src/optimizer/row_group_pruner.cpp

API - immutable objects with range reads. Blob storage exposes a minimal, HTTP-based interface that is largely uniform across providers [27]. Data is uploaded (PUT) and downloaded (GET) as objects: immutable sequences of bytes identified by a key within a namespace called a bucket. Objects cannot be modified in place; however, they can be replaced atomically. This is a property that open table formats such as Apache Iceberg and Delta Lake exploit to build consistent snapshots and transactional semantics on top of object storage [15]. Reads, however, do not need to fetch entire objects: HTTP *range requests* allow reading an arbitrary byte range starting from a given offset within an object. This enables random-access reads on immutable files.

Performance - request size and concurrency (S3). Durner et al. [27] provide a detailed experimental analysis of S3 in the context of analytical query processing, the first in-depth study of this foundational service for OLAP. Their results reveal two findings that determine the performance.

Firstly, request size matters. Every request has a fixed round-trip latency until the first byte arrives (a median of roughly 30 ms on S3, significantly higher than local NVMe), independent of the requested size. For small requests, this round-trip time dominates the total duration, thereby limiting throughput. As the request size grows, the transfer becomes bandwidth-bound rather than latency-bound. However, the bandwidth of a single request is itself capped: per-object bandwidth on S3 ranges from roughly 25 to 95 MiB/s, with a median of 40–50 MiB/s, comparable to a single HDD. Moreover, beyond a certain size, larger requests yield no further benefit: increasing a request from 8 to 16 MiB still delivers minor improvements, but from 16 to 32 MiB, the latency doubles with request size, and the per-request bandwidth limit is reached. Taking both throughput and request cost into account, the study identifies request sizes of 8–16 MiB as cost-throughput optimal for analytical workloads [27].

Secondly, because a single request has a throughput of only tens of MiB/s, saturating the network bandwidth of an EC2 instance requires many concurrent requests. For a 100 Gbit/s instance [4] and with 200–500 simultaneously outstanding requests, a single instance sustains 75–90 Gbit/s from S3, close to its network limit.

S3 latency is not only high but also highly variable. Because S3 is a shared, multi-tenant service, its performance depends on the load from other customers: per-request bandwidth ranges from roughly 25 to 95 MiB/s, with clear daily and weekly patterns [27]. Moreover, a small fraction of requests gives significant tail latency, which can be mitigated by request hedging, i.e., re-issuing requests that exceed a latency threshold [24].

2. BACKGROUND

Pricing model. The pricing structure of blob storage reinforces these design choices. Costs include storage, API (request) costs, and inter-region transfer costs; intra-region traffic between compute instances and the object store is free [6]. For requests within a region, providers charge only by the number of requests, not by their size: downloading a 1 KiB object costs the same as downloading a 1 TiB object, as long as a single HTTP request is issued (on S3, \$0.40 per million GET and \$5.00 per million PUT requests; other vendors price almost identically [27]). Many small requests are therefore wasteful, making per-byte retrieval expensive.

2.4 Differential storage in MotherDuck

MotherDuck, a serverless data warehouse built on the embedded analytical database DuckDB [38], introduces differential storage: a copy-on-write storage layer that brings a single-file, update-in-place database engine like DuckDB to the constraints of cloud storage and multi-user operation [30].

2.4.1 Motivation

DuckDB was designed as an in-process, single-user database, storing the entire database in a single file and modifying it through in-place writes. This design conflicts with the requirements of a cloud data warehouse in two ways. First, in-place modification is incompatible with blob storage: as discussed above, object stores only support immutable objects, so a randomly overwritten database file cannot be hosted on S3 directly and would instead require a file system supporting in-place updates (e.g., network file systems such as Amazon EFS), which is less scalable and more expensive. Second, DuckDB assumes that the underlying database file never changes, which makes it harder for concurrent readers on other hosts to observe a consistent, evolving database safely. Differential storage addresses both problems without modifying DuckDB itself: it is implemented as a userspace file system (FUSE) that stays between DuckDB and the physical storage [30].

2.4.2 Design: immutable snapshot layers

The core idea is to describe the logical database file not as a single mutable file but as an ordered sequence of immutable snapshot layers, each corresponding to a checkpoint and containing only the data written since the previous checkpoint; hence the name “differential”. Incoming writes are handled as follows: the data is instead appended to an append-only active layer buffer, and a mapping from logical byte ranges to physical offsets within the

2.4 Differential storage in MotherDuck

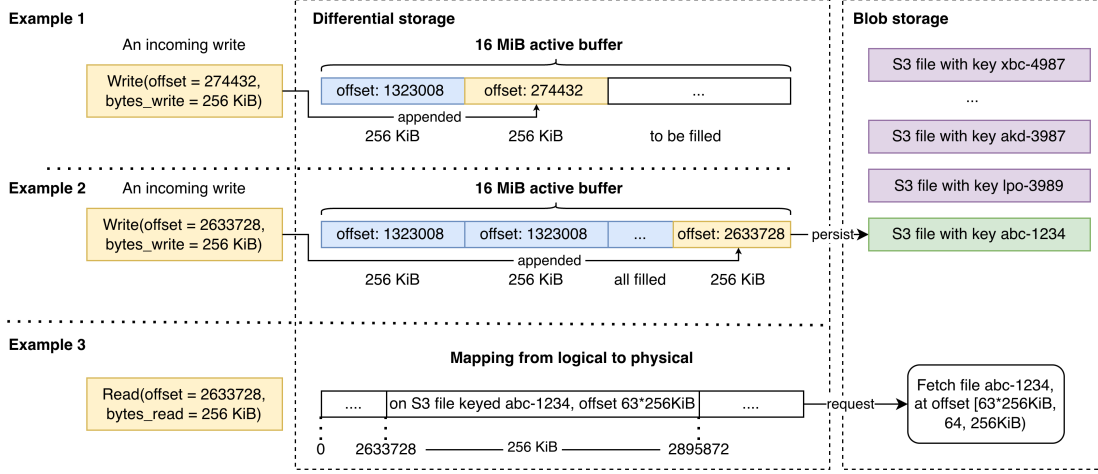


Figure 2.9: Differential storage example. (1) A write is simply an append to an in-memory buffer. (2) A write fills up a buffer, causing a new buffer to be created and the old one to be written to S3. (3) A read consults the mapping information to locate the physical location of a logical address.

layer is maintained. Only one active buffer is open at a time; when it reaches 16 MiB, differential storage persists it to S3. A read of a byte range is split into subranges, each served by the newest layer that contains data for it, since that layer holds the most recent write to those bytes. On a DuckDB checkpoint, the active layer is sealed and becomes the newest immutable snapshot layer; a fresh active layer is opened for subsequent writes, and the layer metadata is recorded transactionally in a separate OLTP database.

An example is shown in Figure 2.9. In **example 1**, DuckDB issues a 256 KiB write at logical offset 274,432; rather than modifying the database file in place, differential storage appends the data to the next free slot in the 16 MiB active buffer and records the logical offset to which it belongs. In **example 2**, a subsequent write to offset 2,633,728 fills the last remaining slot of the active buffer; the buffer, now full, is persisted to blob storage as an immutable object (here with key abc-1234). Note that the buffer may contain multiple entries for the same logical offset (here, 1,323,008), as each overwrite is appended and then resolved to the newest entry. **Example 3** shows the corresponding read path: for a 256 KiB read at logical offset 2,633,728, differential storage consults the logical-to-physical mapping, which indicates that this range resides in object abc-1234 at physical offset 63×256 KiB, and fetches exactly those bytes from S3 with a single range request.

Differential storage follows the same principle that underlies open table formats such as Apache Iceberg and Delta Lake [15], representing database state as immutable files plus

2. BACKGROUND

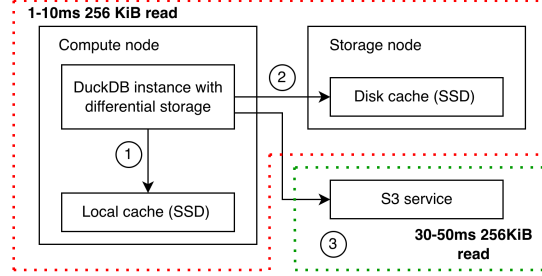


Figure 2.10: Possible caching layers for differential storage. Because layers are immutable, a cached object never needs invalidating, so caching is safe at any of these levels.

versioned metadata, but applies it at the level of byte ranges of a single database file rather than at the level of table data files, thereby making an update-in-place database engine compatible with immutable cloud storage such as blob storage.

2.4.3 Features enabled by differential storage service

Because a snapshot of the database is simply a subsequence of immutable layers, several features that are expensive in conventional architectures reduce to metadata-only operations that duplicate no data [30]. *Snapshotting* is straightforward, as it just means persisting the mapping; *forking* a database is equivalent to applying a new, independent set of mapping layers on top of a shared snapshot, making *zero-copy clones* instantaneous regardless of database size. *Time travel* also comes easily, as it just involves getting and restoring the database with the right snapshots.

2.4.4 Caching

A further consequence of immutability is cacheability: since S3 files never change, they can be cached at any tier without coordination with the writer. We can exploit this by placing multiple layers of cache between DuckDB and S3, all mediated transparently by the differential storage system, as illustrated in Figure 2.10. When DuckDB reads a 256 KiB block of the logical database file, differential storage can serve it from (1) a local SSD cache on the compute node; on a miss, it moves to (2) an SSD-based disk cache on a separate storage node; only when the data is present in neither tier does it (3) fetch the block from S3. The latency difference between these caching tiers is clear: a 256 KiB read served from SSD completes in 1–10 ms, while the same read from S3 takes 30–50 ms.

Throughout this thesis, we study DuckDB without any of these caching tiers, except for the local disk cache, which some of the mechanisms we present use. Our focus is on

2.4 Differential storage in MotherDuck

the cold-read problem: the case where the data is not cached anywhere in the system and must be fetched directly from cloud object storage.

2. BACKGROUND

3

Literature review

In this chapter, we review common file formats for storing analytical data in blob storage, focusing on their suitability for cloud object stores. We do not look into changes to the file formats themselves (e.g., alternative compression schemes, metadata). Instead, we look into the physical layout that each format produces on the object storage service, and how this layout shapes the way data is fetched, that is, the size and number of requests issued. Furthermore, we survey data engines designed to store data in cloud object stores.

3.1 File formats in the cloud

3.1.1 Parquet and ORC

Overview. These are the dominant columnar storage formats in the cloud. They employ the PAX format discussed in Chapter 2. Data is split into row groups, and further within each row group, rows are stored column by column. The PAX-style format facilitates vectorized query processing by the database management system (DBMS) and keeps tuple reconstruction overhead within a row group small [45]. Figure 3.1 describes the layout of the two broadly similar formats. In both, the data of a single column within a row group is stored contiguously: Parquet stores it as a sequence of pages, each starting with a self-describing header. Meanwhile, ORC stores it as a small set of adjacent streams whose locations are derived from the stripe footer. (ORC names its horizontal partition a *stripe*, and uses *row entry* for the 10,000-row index granularity within a stripe; we use row group for the horizontal partition in both formats unless noted.) Contiguity means a column chunk can be fetched from blob storage with a single range request. Writers buffer an entire row group in memory before flushing it to disk. Within a column chunk, data is further segmented into pages (Parquet) or compression chunks (ORC). In Parquet,

3. LITERATURE REVIEW

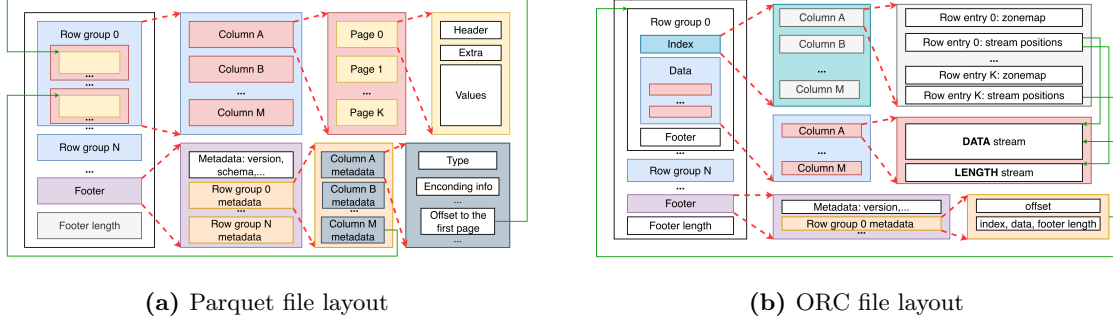


Figure 3.1: Format layout for Parquet and ORC. In principle, both Parquet and ORC employ the PAX-style format, with data split into multiple row groups.

the page is the unit of encoding, compression, and data skipping. ORC decouples these: compression chunks (~ 256 KB) are the compression unit. At the same time, data skipping operates on 10,000 rows per pass, using per-column index entries that record statistics and positions in the streams.

Row group sizes are configured differently across implementations: Parquet writers bound row groups either by tuple count (e.g., 1Mi rows in **Arrow**¹) or by byte size (e.g., 128 MB in **parquet-java**²). In fact, parquet writers are flexible in configuring row group sizes, which vary between writer implementations. Meanwhile, ORC always bounds stripes by byte size. Row-count bounds guarantee a minimum number of tuples per row group for query execution, but let memory consumption during writing grow unboundedly with row width; byte-size bounds cap memory but yield variable row counts. Hybrid approaches are possible. In either case, individual column chunk sizes follow only indirectly, determined by per-column value widths and compression ratios. Hence, the sizes of range requests issued to object storage vary across columns and datasets, making precise control difficult.

How DBMSs adopt Parquet and ORC and how data is read. Database engines can use public libraries (**parquet-java**, **arrow**, **liborc**) or implement native readers tightly coupled to their execution models, as done by DuckDB, ClickHouse, and the Trino library [38; 42; 39].

Regardless of implementation, the read path follows some of the same stages:

- **Metadata bootstrap:** the reader fetches the file metadata, parses the footer, and obtains the schema, offsets, and statistics for all row groups. This metadata can then be cached on the local machine.

¹<https://github.com/apache/arrow/>

²<https://github.com/apache/parquet-java>

- **Scanning and pruning:** projection pushdown selects only the referenced column chunks; predicate pushdown compares query predicates against row-group min-max statistics, optionally against Parquet’s page-level index (supported by Trino or Spark) [43; 42; 14] or ORC’s row-index entries via search arguments (SARGs) [13], and against bloom filters for point predicates, discarding whole row groups, pages before any data is fetched.
- **I/O requests:** the required pages are then issued as range requests; readers can coalesce ranges and issue them asynchronously or synchronously. The row group is also the natural unit of parallelism: Spark splits files at row-group boundaries so that workers scan row groups concurrently.

3.1.2 Future-proof File Format - F3

F3 [45] is a recently proposed file format. An important and relevant aspect of the format is to decouple physical I/O granularity from the row group. The authors argue that the concept of a “row group” in Parquet is overly loaded and tied to the file’s I/O granularity. This is problematic because Parquet’s file writer is not allowed to flush a partial column chunk to disk. This follows from the footer describing each column chunk as a single contiguous offset and length: columns must be written back-to-back, so data for all later columns must be retained in memory while the earlier ones are still being encoded. Therefore, the writer must wait until the entire row group is buffered in memory before writing to disk.

Additionally, the official Parquet documentation and most implementations use column chunks as the unit of I/O during file reading. However, the size of a column chunk can vary drastically depending on the data’s compressibility (e.g., 50 KB vs. 50 MB).

F3 addresses this problem by decoupling the physical I/O granularity from the row group [45]. The row group serves only as a logical partition, a unit of grouping and potentially of data skipping, while the actual on-disk unit is the IOUnit, sized independently at a fixed number (8 MiB by default, the request size shown to be optimal for cloud object storage [27]). An IOUnit belongs to a single column rather than a row group; the writer can flush each column’s IOUnit as soon as it fills, rather than buffering the entire row group in memory as described in Figure 3.2.

The decoupling of the IOUnit from the row group does not fully remove the dependence on row group size. An IOUnit cannot go beyond a row group boundary: when a row group is ingested, every column’s partially filled buffer is flushed, no matter how much

3. LITERATURE REVIEW

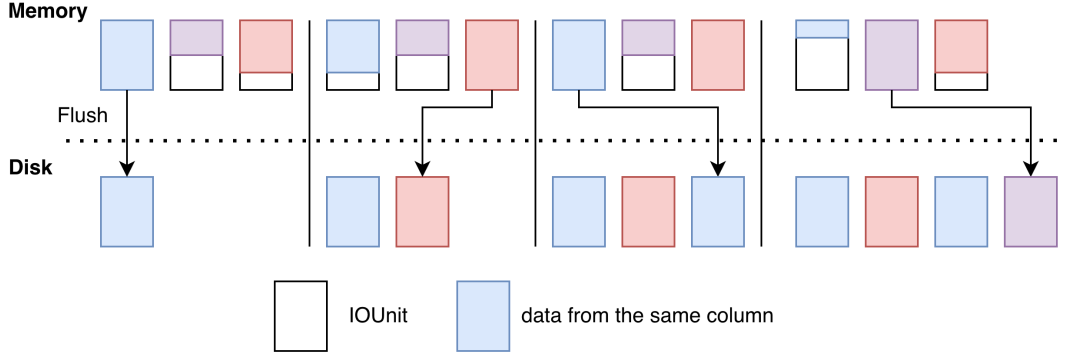


Figure 3.2: Writing data with F3 format. F3 flushes an IOUnit to disk as soon as the column writer buffers enough data in memory. Figure taken from [45]

data it holds. The 8 MiB is therefore only an upper bound. The actual IOUnit size is the minimum of 8 MiB and the compressed size of the column’s data left in the row group. For big columns (e.g., long strings or vector embeddings), the design works as intended: the column chunk is split into uniform 8 MiB units. For small columns, the IOUnit is simply the whole column chunk, and its size again depends on the row count of the row group and how well the data compresses — the same variability the design tries to remove. To get uniform I/O sizes for all columns, the row group must be large enough that even the smallest column reaches 8 MiB of compressed data. The real contribution of F3 is that such large row groups become affordable: write memory grows with the number of columns times the IOUnit size, not with the row group size, so that the row group can grow freely. In Parquet, this is not possible, as the writer must buffer the entire row group. The problem of varying I/O sizes is thus not solved but moved to very large row groups, which in turn comes at the cost of coarser data skipping at the row group level (however, partially solved via page index [14]).

The more novel part of F3 is the incorporation of the decoder, the code needed to read the data, into the file itself. As a result, compatibility becomes far less of an obstacle, enabling developers and researchers to explore new encoding strategies without being constrained by reader support. This aspect, however, falls outside the scope of this thesis.

3.2 Pixels: a columnar store for the cloud

The study [18] sheds light on how data should be organized in cloud object storage to maximize retrieval speed and throughput. We present some relevant findings from Pixels

in this section.

3.2.1 PAX or DSM

PAX is beneficial over DSM for the following reasons [18]:

- Self-descriptive row groups: a row group has its own metadata and contains the full data of every row inside it. As a result, database engines can parallelize table scanning on a per-row-group basis, i.e., each thread handles one row group independently without complex coordination. Tuple reconstruction also never crosses a row group boundary. Pruning benefits as well: since all columns share the same horizontal boundary, one statistics check skips the entire row group across all columns at once.
- Cache-friendly: PAX was originally designed as an in-page layout [2], where the data for each column on a page sits next to the others. Scanning one column then does not drag the bytes of other columns through the CPU cache.
- Easier data ingestion: compared to DSM, where the writer must either buffer the whole table or keep one open object per column, a PAX writer only buffers one row group and flushes it as soon as it is filled, without complicated coordination.

This aligns with the arguments made for the F3 file format. Its authors reason that F3 can further reduce the remaining ingestion cost of PAX of buffering the entire row group with fixed-size physical units called IOUnits, as discussed in the previous section. Splitting data into row groups is also easier for catalog and metadata management. Organizing data as DSM would make coordination harder, since each column is segmented independently. Pruning is also complicated with DSM: statistics exist per column chunk, but chunks of different columns do not share boundaries, so skipping requires intersecting row ranges. However, DSM has one advantage over PAX: it is not limited by row group boundaries, so each column is stored as a single contiguous sequence, and the problem of varying column chunk sizes naturally disappears. Finally, PAX treats cache locality as its primary goal, but this matters less for remote storage: read latency already dominates, which significantly reduces the impact of poor CPU cache locality. In the cloud setting, the layout choice is instead driven by request granularity, pruning, and parallelism. PAX is still the design choice of Pixels.

3. LITERATURE REVIEW

3.2.2 Request size and reading method

Pixels makes two important observations about reading data from object stores. When it comes to S3-like services, hiding and mitigating latency are among the biggest challenges. Pixels reasons that we can solve this in two ways: (1) increasing request sizes, (2) increasing concurrency.

Reducing the number of requests or increasing request sizes. Every GET request to S3 has a first-byte latency of tens of milliseconds, so fetching one large request instead of small ones reduces this fixed cost over more bytes; the latency is paid once rather than many times. Fewer, larger requests therefore shift the workload from latency-bound to bandwidth-bound, which is suitable for object storage. This is confirmed in [27].

Pixels does this by optimizing the storage layout to increase the chance of coalescing. Pixels reorders the physical layout to bring co-accessed data close together, in two steps as illustrated in Figure 3.3. First, **intra-row-group column ordering** puts data of frequently accessed columns next to each other; finding the best order is NP-hard, so Pixels uses a simulated-annealing algorithm from a prior work on HDFS [19]. Second, since a scan typically touches many row groups, the chunks of the same column are scattered across them and cannot be merged; **inter-row-group column compaction** compacts a fixed number of row groups (32 by default) into one large object, placing the chunks of the same column contiguously while keeping the row groups as logical units with rewritten footer offsets. A scan of a few columns over the compacted layout then needs one request per column instead of one per column per row group.

Increasing concurrency or the number of concurrent requests. An in-flight request barely occupies the compute instance: first-byte latency is idle waiting, and the transfer is offloaded to the NIC, which places data directly into memory. Many requests can therefore proceed concurrently at negligible cost, overlapping their latencies and utilizing available bandwidth. Pixels achieves this with an adaptive reading method: requests below a threshold (1 MB by default) are sent via the asynchronous, non-blocking S3 client, which allows a scan task to keep many small, latency-bound requests in flight, while larger, bandwidth-bound requests use the synchronous client [18]. Pixels also applied a rate limit to curb the number of concurrent requests during a burst.

3.3 AnyBlob: exploiting cloud object stores for OLAP workloads

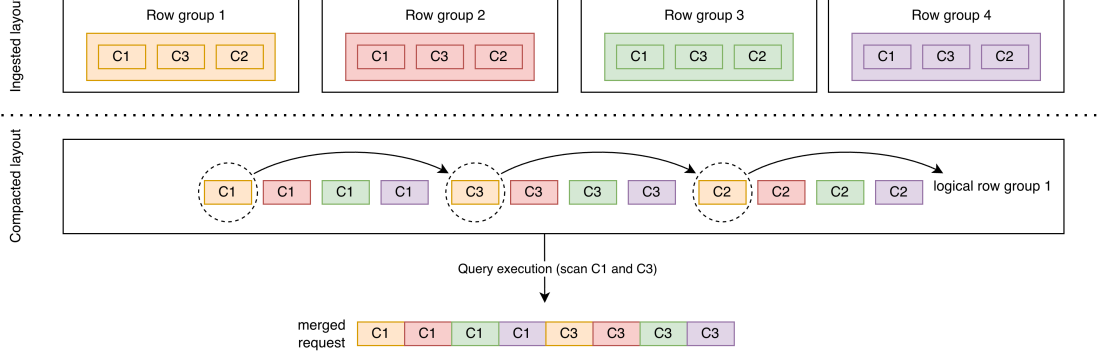


Figure 3.3: Illustration of intra-row-group column ordering and inter-row-group column compaction in Pixels. Persisted data does not follow the original ordering (C1, C2, C3) but rather (C1, C3, C2), as the pair C1 and C3 is usually co-accessed. The compaction logic moves column data from other row groups into a single row group, preserving the column order. The result is that one merged request across columns can cover many row groups. Figure taken from [18]

3.2.3 Discussion

In the evaluation, Pixels outperforms Parquet by more than an order of magnitude on a real-world wide-table workload and by a factor of 1.93 on TPC-H. These gains can be attributed to two factors: (1) larger requests, enabled by the layout optimization and request merging, hiding the first-byte latency over more bytes; (2) a high level of concurrency, enabled by asynchronous reading, overlapping the latencies of the remaining requests and utilizing the available bandwidth effectively.

3.3 AnyBlob: exploiting cloud object stores for OLAP workloads

AnyBlob [27] further investigates how to make cloud object stores more suitable for OLAP workloads by characterizing them across aspects such as latency, throughput, and costs, and by quantifying what is best for blob storage, i.e., how to read from the S3 service to utilize resources on modern cloud hardware.

3.3.1 Characterizing cloud object storage

Cost structure. The cost of S3 is discussed in Chapter 2. Other storage options are neither as scalable nor as elastic as cloud object stores, and are often more expensive. Amazon Elastic Block Store [7] enables users to attach a data volume to AWS EC2 instances.

3. LITERATURE REVIEW

Table 3.1: Comparison of AWS storage services derived from [7; 3; 4; 12]

Storage service	Price (\$/TiB)	Through-put	Scalability	Durability	Latency ^a
EBS io2 Block Express	128.00	32 Gbps	4 GB–64 TB	99.999%	sub-ms
EBS gp3 (SSD)	81.92	16 Gbps	1 GB–64 TB	99.8–99.9%	single-digit ms
EBS st1 (HDD)	46.08	4 Gbps	125 GB–16 TB	99.8–99.9%	no guarantee
NVMe (instance store)	158.40	16 Gbps	1.8 TB	instance lifetime ^b	not provided
S3 Standard	23.55	5–400 Gbps ^c	unlimited	99.999999999%	30–50 ms ^d

^a Per block read; EBS has 16 KB I/O size. ^b Data persists only for the lifetime of the instance.

^c Depends on instance network bandwidth. ^d For a 256 KB request.

Size	Total (Cold) [ms]					Total (Hot) [ms]				
	p5	p25	p50	p75	p95	p5	p25	p50	p75	p95
1 KiB	14	25	32	41	55	12	15	27	41	68
1 MiB	68	88	96	107	140	43	60	80	97	125
4 MiB	125	175	220	270	365	55	113	160	210	300
8 MiB	130	185	225	265	360	90	130	170	230	310
16 MiB	220	350	395	475	590	175	215	300	375	490
32 MiB	610	760	840	950	1110	540	740	850	970	1160

Table 3.2: Request latencies on AWS S3 (c5n.large, eu-central-1), read from Figure 2 of [27]. Moving from 8 MiB to 16 MiB doubles the size, but also almost doubles the latency, which means we reach the bandwidth cap for an object.

It has different flavors for different types of disks. Table 3.1 compares different storage services. For example, General Purpose SSD (gp3) costs \$81.92/TiB compared to S3’s \$23.55/TiB. HDD-backed disks are cheaper; however, they still cost twice as much as S3 and offer limited bandwidth. Surprisingly, local NVMe disks on AWS instances have a bandwidth of only 16 Gbit/s, well under what S3 can achieve. Therefore, S3 remains the best option for storage, as it offers the cheapest storage option, with durability and comparable throughput.

Latency We took the results from AnyBlob, shown in Table 3.2. It is evident from the results that, for small request sizes, first-byte latency dominates overall latency, indicating

3.3 AnyBlob: exploiting cloud object stores for OLAP workloads

that throughput is limited by it. Pushing the request size reduces the effect of round-trip latency; however, the benefit is capped at 8–16 MiB. Moving from 8 MiB to 16 MiB doubles the size, but also almost doubles the latency, which means we reach the bandwidth cap for an object.

An important characteristic of S3 is that its latency is noisy, because the S3 service is shared across a cloud environment and among many customers. Performance fluctuates throughout the day, depending on the system’s state.

Throughput. AnyBlob [27] experimented by downloading multiple 16 MiB objects within a 1-second window on a 100 Gbit/s instance. The result showed that they can reach 75–90 Gbit/s with 8–16 MiB requests, but using 256 KiB is capped at 23 Gbit/s. This means that at an appropriate level of concurrency, request granularity, and high bandwidth, retrieval throughput can reach the network bandwidth.

Optimal request size. Request size is an important design decision, as it affects both performance and cost: providers charge per request, so larger requests are cheaper for the same data volume. Throughput improves with size until the bandwidth limit of a single request is reached. At the same time, requests should stay small enough that even small tables can be split across parallel downloads. Based on this trade-off, the paper classifies 8–16 MiB as the cost-throughput optimal request size for OLAP.

Quantifying cloud storage retrieval. Durner et al. [27] propose a model that quantifies the level of concurrency required to saturate the available network bandwidth for a given request size. A single request of a given size completes in a fixed base latency plus a size-proportional transfer time, and thus delivers a limited per-request bandwidth on its own. The number of concurrent requests needed to reach a target throughput follows as:

$$\text{requests} = \text{throughput} \cdot \frac{\text{baseLatency} + \text{size} \cdot \text{dataLatency}}{\text{size}} \quad (3.1)$$

For S3 reads with sufficiently large request sizes, the authors measure a median base latency of 30 ms and a median data latency of 20 ms/MiB, which gives 40–50 MiB/s per request. Saturating a high-bandwidth instance therefore requires hundreds of requests in flight. Table 3.3 shows the estimated required concurrency for cost-optimal request sizes at two typical instance bandwidths. Doubling the request size from 8 to 16 MiB barely reduces the required concurrency, which is exactly why the cost-throughput optimal range ends around 16 MiB. Meanwhile, a large number of concurrent requests are required if the

3. LITERATURE REVIEW

Request size	Per-request bandwidth	Required concurrent requests	
		50 Gbit/s	100 Gbit/s
256 KiB	~7 MiB/s	~835	~1,670
8 MiB	~42 MiB/s	~142	~283
16 MiB	~46 MiB/s	~130	~261

Table 3.3: Concurrent requests required to saturate a given network bandwidth on S3, following the model of [27] with median base latency ~30 ms and data latency ~20 ms/MiB.

request size remains small (e.g., 256 KiB). Small requests significantly inflate the required concurrency: at 256 KiB, the model estimates roughly 1,700 requests in flight are needed to saturate 100 Gbit/s. Within this scheme, the estimate should be read as a lower bound, and likely an optimistic one: the model accounts only for request latency, while in practice such high concurrency runs into additional limits that cap the achievable bandwidth, e.g., S3’s per-prefix request-rate limits, connection management overhead, per-request CPU cost for network handling and request signing, and request scheduling and buffering inside the database engine. These client-side costs, which grow with the number of requests are the overheads AnyBlob’s `io_uring`-based design targets [27].

3.3.2 Efficient I/O engine

To sustain hundreds of in-flight requests without oversubscribing threads, AnyBlob implements its own I/O engine on top of `io_uring`, a Linux kernel interface based on lock-free submission and completion queues shared between user and kernel space, which reduces the number and cost of system calls. Operations from multiple tasks are batched before submission, further reducing system call overhead. As a result, a small number of scheduler threads saturate the network at a fraction of the CPU cost of thread-per-request designs or vendor SDKs [27]. The evaluation shows that the `io_uring` implementation can reach the throughput of the vendor SDKs while using 30% less CPU.

3.3.3 Anyblob and integration

Compared to Pixels, AnyBlob’s storage format in Umbra takes a different approach. Instead of merging small requests at read time, the data is organized at write time so that a single-column chunk directly corresponds to a well-sized request. The table is stored as immutable, column-major blocks, and the target size of each column chunk within a block is the cost-throughput-optimal request size of 16 MiB, derived in their study and

3.4 Putting it all together

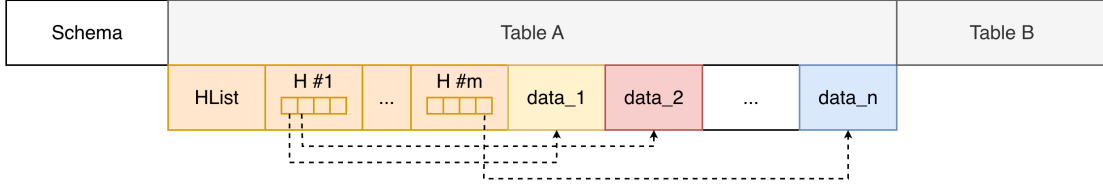


Figure 3.4: Illustration of the object structure in AnyBlob. Each coloured shape is an object in S3. The header list can be replaced atomically, which guarantees consistency for the database. Plus, AnyBlob tunes each data block during ingest so that its column chunks land at a size S3 serves well. Figure taken from [27].

inspired by [31]. The structure on cloud object storage is shown in Figure 3.4. Metadata is stored on object storage as follows. Under the database prefix, Umbra stores the schema information, which contains everything needed to read the database. Each table has its own subprefix containing a header list object, and the header objects themselves. The header objects in turn point to the data block objects that hold the data. Because objects can be updated atomically, replacing the header list creates a consistent snapshot [27]. Umbra uses the tuple count as a tuning knob: during data block writing, the tuple count is adaptively picked so that no encoded column chunk falls below 2 MiB, thereby keeping the number of requests small, while chunks that exceed 16 MiB are fetched via multiple smaller range requests, keeping download latencies uniform across columns.

3.3.4 Discussion

This work reiterates what is important to get the best out of the S3 service, which has already been stated in [18], but provides a deeper understanding of how to do that. We learned that greater read granularity is key to cost-throughput efficiency, while high concurrency is required to saturate the available network bandwidth. AnyBlob also proposes an alternative system to handle S3 requests efficiently, based on `io_uring`, rather than using the well-maintained, AWS-supported SDK.

3.4 Putting it all together

Across the literature we have studied, we summarize the key findings in using blob storage for OLAP workloads.

Concurrency and request size are the deciding factors for the full utilization of S3. S3 favors requests of a certain size range (8–16 MiB is cost-throughput optimal [27]).

3. LITERATURE REVIEW

Since S3 has a high latency compared to local storage while offering high aggregate bandwidth, many concurrent requests are needed, on the order of hundreds to saturate a 100 Gbit/s instance.

Data layout decides the request size. The layout determines whether well-sized requests are possible. F3 and the storage format of AnyBlob/Umbra shape the data at write time: F3 caps the physical unit with a fixed-size IOUnit (8 MiB), while Umbra tunes the tuple count of each block so that column chunks land in the optimal request range [45; 27]. Pixels instead improves the layout after ingestion: intra-row-group column ordering places co-accessed columns next to each other, and inter-row-group column compaction merges tens of row groups into one large object so that chunks of the same column become adjacent, allowing the reader to merge many small chunk requests into a few large ones [18]. All of these approaches address the same root problem: the row group boundary, combined with different data types and compression ratios per column, produces column chunks of widely varying sizes within one row group, leading to inefficient request sizes, and each system removes this variance in a different way and at a different point in the data’s lifetime.

Asynchronous I/O is key to increasing concurrency. With synchronous I/O, the number of outstanding requests is capped by the number of threads, which is far below the hundreds of requests required. Asynchronous I/O decouples the two and also utilizes idle CPU time to execute queries. In Pixels, this is done through an I/O scheduler that batches chunk requests for a scan task, merges nearby requests, and sends small requests via the non-blocking, asynchronous S3 client, while large requests use the synchronous client [18]. In AnyBlob, a small number of threads each manage many connections through `io_uring`-based asynchronous system calls, with each request driven by a state machine, reaching the required concurrency without thread oversubscription and at a fraction of the CPU cost of the vendor SDKs [27].

4

Motivation and design

In this chapter, we discuss and characterize the current state of DuckDB and MotherDuck on cloud object storage, focusing on how they physically organize data on S3. Based on this analysis, we motivate our approach and describe the design that bridges the gap between executing DuckDB on a local machine and executing DuckDB in the cloud.

4.1 Revisiting object storage performance

Before analyzing the access patterns, we introduce a baseline for what AWS S3 itself can deliver in our environment, and compare it against the results reported by AnyBlob [27]. AnyBlob tied its study to a tailored I/O engine built on the low-overhead asynchronous interface `io_uring` (Chapter 3). While such an engine reduces the CPU cost of high request concurrency, the contributions of this thesis, data layout and prefetching, are orthogonal to the choice of I/O engine. We therefore keep issue all S3 requests through the official AWS SDK, for three reasons: (1) it is production-ready, well-maintained, and natively supported by AWS; (2) it is the industry standard, so our results reflect what a typical deployment would observe; and (3) as we verify below, when used with AWS’s recommended best practices it achieves throughput sufficient for our purposes. The goal of this section is thus to confirm that the request size and concurrency characteristics identified by AnyBlob [27], 8–16 MiB requests and hundreds of concurrent requests, hold when reading from S3 through the standard SDK, so that we can rely on them as design parameters in the rest of this chapter.

We created thousands of objects for each of the sizes 256 KiB, 1 MiB, 2 MiB, 8 MiB, 16 MiB, and 32 MiB, and then measured latency and throughput at different concurrency levels. All runs used **AWS Common Runtime (CRT)** library [10] to interact with the

4. MOTIVATION AND DESIGN

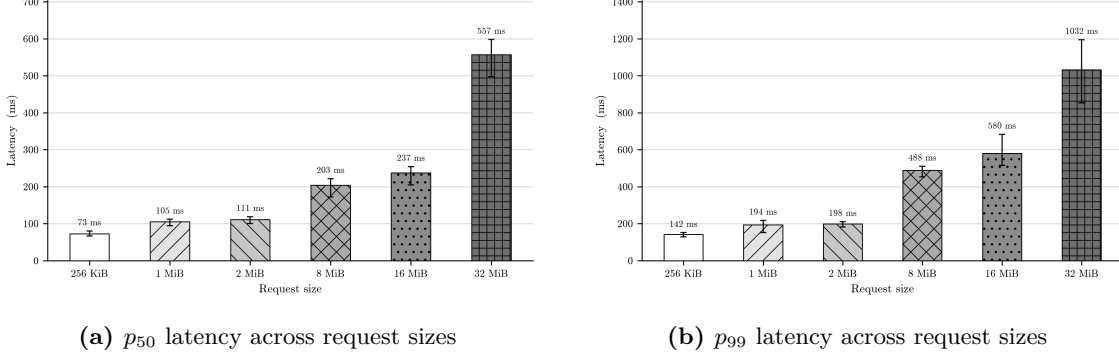


Figure 4.1: p_{50} and p_{99} latency for S3 requests across different sizes (**us-east-1, m6idn.16xlarge** instance). The whiskers show the min and max across runs; the bar height is the mean across runs. Doubling the request size from 16 MiB to 32 MiB more than doubles the latency, indicating the per-object bandwidth limit is reached.

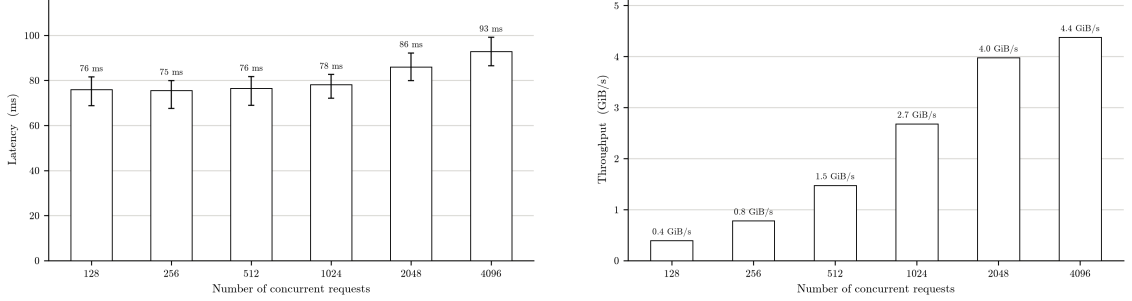
S3 service, as this library is designed to work on high-bandwidth Amazon EC2 instances. The instance used is **m6idn.16xlarge** [5], which offers 64 vCPUs, 256 GiB of memory, and 100 Gbps of network bandwidth.

4.1.1 Latency

First, we issued requests one at a time, varying their sizes, and recorded the p_{50} and p_{99} latency values. The results are shown in Figure 4.1. Compared to AnyBlob [27], the latency for 8 MiB and 16 MiB requests is lower; however, the overall trend remains the same. The latency for smaller request sizes (256 KiB–2 MiB) matches the results in [27]. At these sizes, latency is dominated by the round-trip time, while at larger request sizes (8 MiB–32 MiB), it is bounded by the per-object bandwidth. Doubling the request size from 16 MiB to 32 MiB results in $2.4\times$ higher latency, indicating that the per-object bandwidth limit is reached and that further increasing the request size does not improve retrieval performance.

An important finding of AnyBlob [27] is that S3 latency is noisy and varies over time. We observed the same behavior in our benchmark, especially for larger request sizes. The p_{99} latency (Figure 4.1b) can be twice the p_{50} latency, so our design must account for this variability. For example, a 32 MiB request can take 1.1 s, significantly affecting query execution time. An approach to alleviate this, suggested by both Pixels [18] and AnyBlob [27], is request hedging (or request retry), in which slow or unresponsive requests are re-issued. This is also recommended by cloud providers [11].

4.1 Revisiting object storage performance



(a) p_{50} latency at different numbers of concurrent 256-KiB requests.

(b) Throughput when querying at 256 KiB at different concurrency levels.

Figure 4.2: p_{50} latency and throughput for S3 requests across different concurrency levels and using 256 KiB requests (**us-east-1, m6idn.16xlarge** instance). Doubling the concurrency from 1024 to 2048 yields a 48% throughput increase, and going further from 2048 to 4096 gives roughly 10% (4.0 GiB/s to 4.4 GiB/s). A concurrency level of about 1024-2048 therefore offers the best trade-off for 256 KiB requests.

4.1.2 Request sizes and throughput

We first examine the maximum throughput achievable with 256 KiB requests by varying the number of concurrent requests. This request size is important because it is the granularity at which DuckDB issues requests to its I/O subsystem. At a concurrency of 1, each 256 KiB request has a median latency of 73 ms, corresponding to a per-request throughput of roughly 3.4 MiB/s. In an ideal scenario, if this latency could be sustained at high concurrency, saturating a network bandwidth of 100 Gbit/s (11.6 GiB/s) would require 11.6 GiB/s/3.4 MiB/s, or roughly 3,500 concurrent requests. In practice, however, latency degrades under load: increasing concurrency from 1024 to 4096 raises the median latency from 78 ms to 93 ms. Doubling the concurrency from 1024 to 2048 yields only a 48% increase in throughput, and pushing it further from 2048 to 4096 yields a small improvement of roughly 10% (4.0 GiB/s to 4.4 GiB/s). We conclude that 1024-2048 is the optimal spot for 256 KiB requests. Throughput does not scale simply with the number of concurrent requests, as sustaining a large number of in-flight requests incurs additional overhead; in a later section, we therefore use a thread pool of 1024–2048 threads to issue 1024–2048 concurrent requests. Another factor that could affect this scaling is AWS-side request-rate scaling, for example the documented limit of 5,500 GET requests per second per prefix. This is hard to quantify, as AWS does not document precisely how request rates are scaled beyond or within these limits [11].

Next, we look at the throughput other request sizes can achieve across different concur-

4. MOTIVATION AND DESIGN

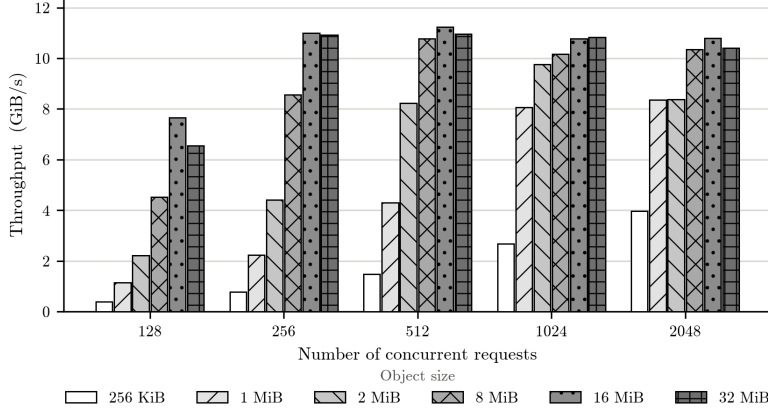


Figure 4.3: Throughput at different request sizes (**us-east-1, m6idn.16xlarge** instance). With a request size of 8 MiB or 16 MiB, only 256-512 concurrent requests are needed to saturate the network bandwidth.

rency levels. Figure 4.3 shows the results. Using 256 KiB requests, we can only reach 4 GiB/s at 2048 concurrent requests, while with 1 MiB and 2 MiB requests, we achieve more than double that (8–10 GiB/s) with 1024 concurrent requests. Request sizes of 8 MiB to 16 MiB remain the sweet spot: they achieve around 11 GiB/s and reach this with only 256 to 512 concurrent requests. This confirms that we should target 8–16 MiB requests, as they can deliver optimal throughput while keeping concurrency low.

4.1.3 Implications for the design

256 KiB requests cannot saturate the network. If we keep DuckDB’s current 256 KiB read granularity, sustaining maximum throughput during table scans would require around 1024–2048 concurrent requests to S3. Even then, 256 KiB requests top out at roughly 4 GiB/s, about a third of the 11 GiB/s that larger requests achieve on the same instance.

S3 favors 8 MiB to 16 MiB requests. In terms of both latency and throughput, S3 offers the best performance at these request sizes while keeping the level of concurrency reasonable. Hence, when bringing DuckDB to S3, we should aim to make I/O requests larger while maintaining a few hundred concurrent requests.

Tail latency must be handled. S3 latency is noisy: the p_{99} latency can be twice the p_{50} , and a single slow request can stall an otherwise well-pipelined scan. Larger requests make this worse, as more data is at stake per request.

4.2 An analysis of differential storage layout on S3 - TPC-H, TPC-DS, and a MotherDuck workload

4.2.1 Methodology

As described in Chapter 2, differential storage provides a standard file system interface on top of the S3 service, through which DuckDB transparently writes and reads. It does so by appending incoming writes to an active in-memory buffer and flushing when it is full (16 MiB), while maintaining a mapping from the logical position (in DuckDB’s single-file format) to the physical position on S3, i.e., which S3 object and at which offset within that object each logical block lives. With the current behavior, differential storage maintains a single active buffer: whatever blocks DuckDB flushes are appended to the same buffer and end up in the same S3 object, in flush order.

To study how data is laid out on S3, we prototyped a minimal version of differential storage that records this mapping. Combining the mapping with the block metadata exposed by `pragma_storage_info`, we can break down every S3 object into 256 KiB blocks and, for each block, detail its contents down to the granularity of `(table, column, row group)`.

Figure 4.4 illustrates the write path. DuckDB writes at logical position $12 \text{ KiB} + 256 \text{ KiB} \times 8$; we append this to the active buffer and record that the range $[12 \text{ KiB} + 256 \text{ KiB} \times 8, 12 \text{ KiB} + 256 \text{ KiB} \times 9)$ lives in the buffer at offset $256 \text{ KiB} \times 62$. DuckDB then writes at $12 \text{ KiB} + 256 \text{ KiB} \times 17$; we append it and record the range $[12 \text{ KiB} + 256 \text{ KiB} \times 17, 12 \text{ KiB} + 256 \text{ KiB} \times 18)$ at buffer offset $256 \text{ KiB} \times 63$. Once the buffer is full, it is persisted as a single S3 object, and a fresh buffer is started.

The analysis in this section uses this default write path, ingesting TPC-H and TPC-DS at SF=500 with multiple threads. We report the default DuckDB row group size ($\times 1 = 122,880$ rows) and, where relevant, other variants. We aim to motivate approaches to a better layout, one that helps the read path make larger requests, i.e., coalescing.

4.2.2 TPC-H and TPC-DS

Mixed data on S3 Because DuckDB writes at 256 KiB block granularity and differential storage packs 64 such blocks into each 16 MiB object, an S3 object contains many blocks, each holding data from different columns. With a single shared write buffer, the blocks flushed in one time window, across all columns and all worker threads, are interleaved into the same object. Figure 4.5 plots the distribution of distinct `(table, column)` pairs per

4. MOTIVATION AND DESIGN

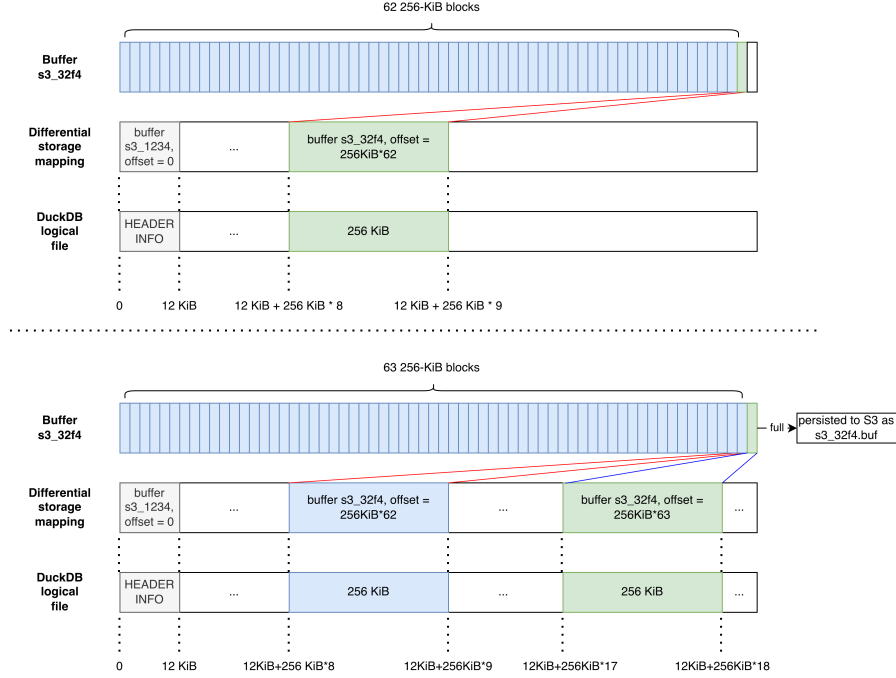


Figure 4.4: Illustration of differential storage write path - how the mapping is updated and how buffer is persisted to S3 when it is full.

S3 object: at the median, an object holds **16** distinct columns on TPC-H (essentially all of `lineitem`'s 16 columns) and **24** on TPC-DS (up to 139), and almost no object holds a single column: 1.3% on TPC-H and none on TPC-DS. Figure 4.6 shows this directly: ten sampled objects per dataset, each rendered as a box of stacked 256 KiB blocks colored by (table, column).

Logical contiguity Because the single active buffer appends blocks in flush order, logically adjacent blocks do tend to stay physically adjacent: under the default single-buffer writer, 98% of consecutive logical block pairs land in the same S3 object. This logical contiguity, however, is not what a query cares about. A scan does not read logically consecutive blocks; it reads the blocks holding the columns it touches, and coalescing can happen only when those blocks, the data read together in one scan, stay physically close. What matters is semantic locality, not raw block-order contiguity, and this is exactly what the single buffer lacks. Each 16 MiB object mixes blocks from many columns, so any one column's blocks are distributed across objects. A column scan therefore cannot coalesce its reads. Differential storage is not bound to this layout: it lets us place blocks freely across S3 objects, and we are not restricted to a single active buffer. By keeping multiple

4.2 An analysis of differential storage layout on S3 - TPC-H, TPC-DS, and a MotherDuck workload

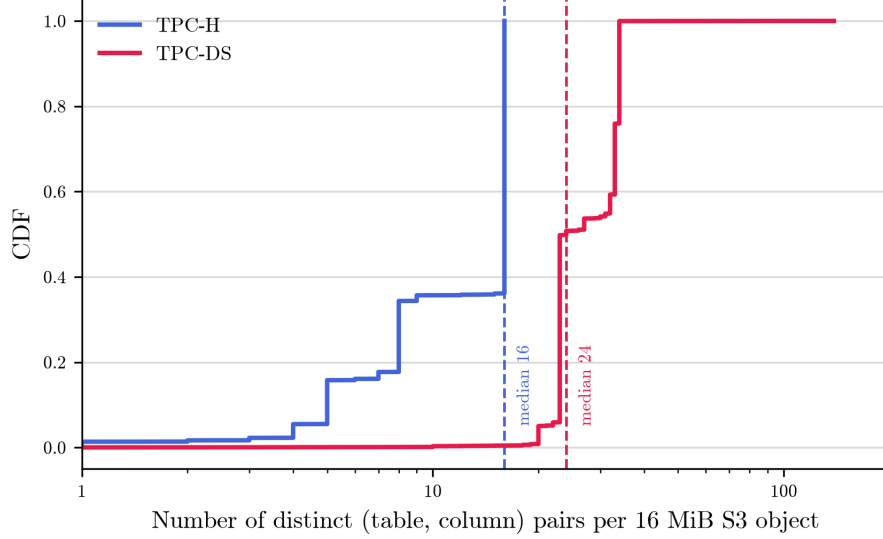


Figure 4.5: Cumulative distribution of the number of distinct (`table`, `column`) pairs residing on one 16 MiB S3 object under differential storage’s default writer (`single_buffer`) at SF 500. Almost no 16 MiB object holds only data from a single column: the median is 16 columns per object on TPC-H and 24 on TPC-DS. A scan touching a few columns cannot coalesce or cache whole files, since those contain unneeded columns.

active in-memory buffers, one per column, we group each column’s blocks so that the data a query reads in a single scan is physically contiguous and coalesces into large requests, giving up the whole-file sequential adjacency that the single buffer happens to preserve.

Optimistic caching suffers too. When reading a block, if we also fetch and cache the whole S3 object containing it, it becomes less efficient because we cache data the query will not touch. This suggests intercepting the write path: passing down context so that we know what data we are writing, and maintaining multiple buffers that host related data instead of a single one.

We prototyped this idea by passing write context down to the file system and then maintaining per-column buffers. When a block write comes in, we put that block into the buffer corresponding to that column. The detailed implementation will be discussed in Section 4.3.3. The high-level idea is to co-locate data from the same column, so that we can either coalesce the reads or cache the whole object, instead of reading only one 256 KiB block. The result of the layout is sampled in Figure 4.7: under the per-column buffer, almost every 16 MiB object is filled by a single column (one solid color per box), in contrast to the interleaving of the default writer in Figure 4.6.

4. MOTIVATION AND DESIGN

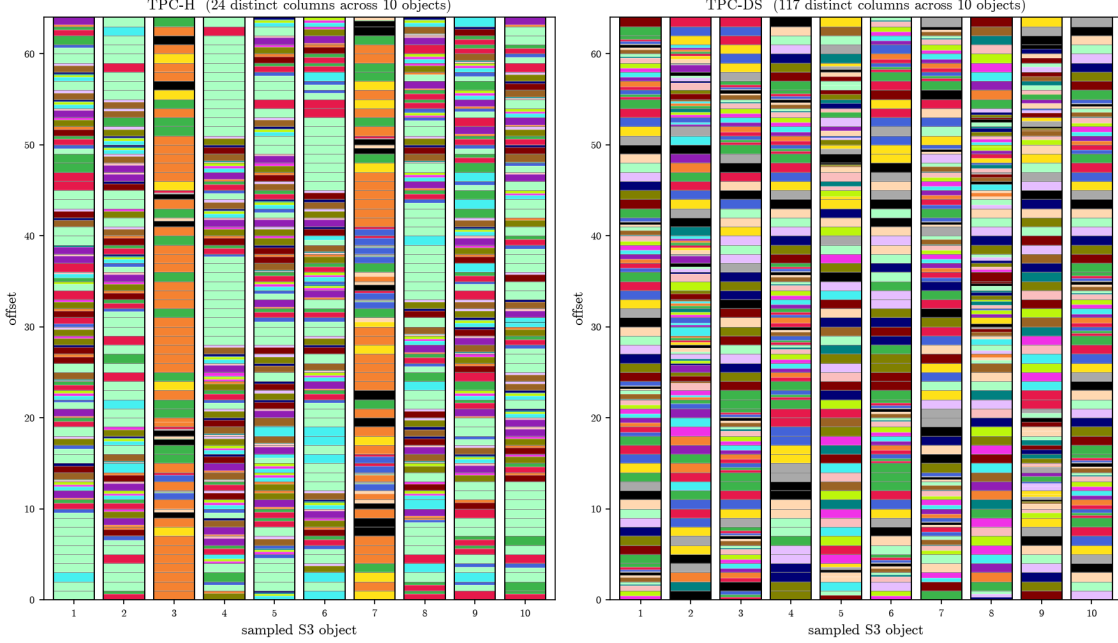


Figure 4.6: Ten sampled 16 MiB S3 objects per benchmark. Each box is one object: a stack of 64 blocks of 256 KiB (offset 0 at the bottom); within a block, each (table, column) segment is a slice sized by the bytes it occupies and colored by its (table, column). S3 objects generally contain a mix of data from multiple columns, no object is dominated by any one column.

Row group order We now discuss how row-group order affects the read path. DuckDB distributes row groups to worker threads roughly sequentially, so the order in which row groups are laid out across S3 objects has a large impact on how data is retrieved. If consecutive row groups are co-located in the same object, handing them out sequentially makes all threads read from the same object at once. This is good for coalescing; the co-located blocks a thread reads now merge into one large request. However, it is a problem for caching. All threads hit the same object at once and then move on together, so prefetching the rest of an object into the cache to serve later reads gives no benefit; the data is consumed immediately by the threads already on the object, not reused later. This motivates spreading the work so that a thread fetches data that other threads are not using at the same time but will use later. One intuitive way is to schedule row groups with a stride rather than strictly sequentially, so that row groups close together on S3 are accessed at different times.

Consider a cache that, on reading one 256 KiB block, stores the whole containing S3 object to local disk. With a sequential layout (row groups 1–4 co-located in one object), at

4.2 An analysis of differential storage layout on S3 - TPC-H, TPC-DS, and a MotherDuck workload

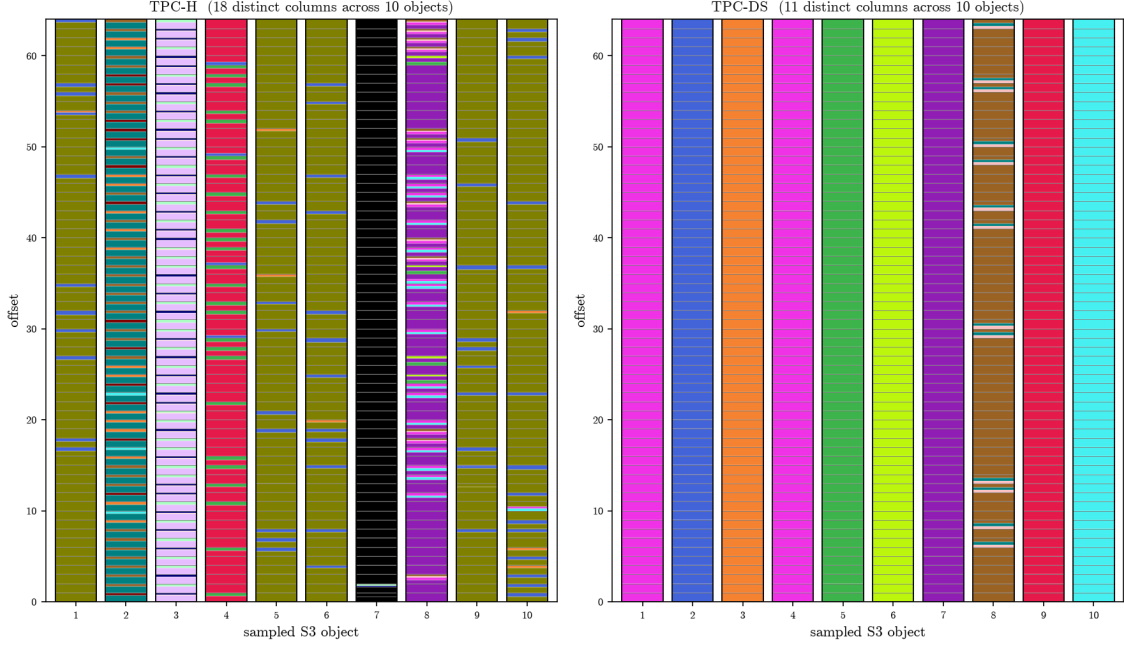


Figure 4.7: Ten sampled 16 MiB S3 objects per benchmark under the prototype **per-column** write path. Each box is now dominated by a single *(table, column)*: co-locating a column’s blocks in the same object is what lets a scan coalesce its reads into large requests, or cache the whole object usefully, instead of fetching isolated 256 KiB blocks.

time T_0 the four threads read row groups 1, 2, 3, 4: they issue individual 256 KiB GETs, and the object is cached, but as the threads move on to row groups 5–8 the cached object is already useless. With an interleaved layout, the cache pays off: at T_0 the threads read row groups 1, 5, 9, 13 (four different objects, all now cached), and at T_1 they read 2, 6, 10, 14, which are served from cache.

Figure 4.8 illustrates the sequential order of row group IDs: each subfigure samples S3 objects holding one column (sorted by the smallest row-group ID in the object) and plots the row group IDs each object contains; a tight rising staircase means the object holds a contiguous, ordered run of row groups. However, optimistic writing can break and add noise to this sequential order in different ways:

- **Parallel writing with imbalanced ingest times.** We would expect a column’s blocks to be emitted in row group order and thus clustered within each object, since threads are handed consecutive row groups. But when threads finish row groups at different times, e.g., because columns compress at different rates, or because of jitter in the system, this order changes. Table 4.1 quantifies this across some representative columns: every object still holds a near-contiguous run.

4. MOTIVATION AND DESIGN

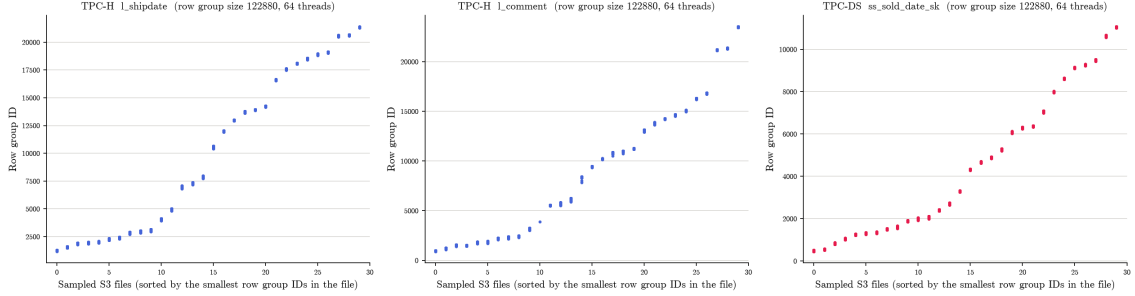


Figure 4.8: Row group layout under the `per-column` writer. Each subfigure samples 30 S3 objects holding one column (x-axis, sorted by the smallest row group ID in the object) and plots the row group IDs the object contains (y-axis), all at row-group size 122880 with 64 threads. Left: TPC-H `l_shipdate`. Middle: TPC-H `l_comment`. Right: TPC-DS `ss_sold_date_sk`. A tight rising staircase means the object holds a contiguous, ordered run of row groups, which makes coalescing data blocks of consecutive row groups possible.

- **Partial blocks.** Partially-filled blocks add some disorder: residues that do not fill a 256 KiB block are held until the end of the ingest and merged out of order. This is a minor effect; instrumenting the ingest shows that fewer than 0.5% of blocks are flushed as end-of-ingest residue, so it is not the main driver.
- **Concurrent optimistic writers.** Each thread maintains its own `PartialBlockManager`; it does not receive row groups in sequential order, so it can mix data from very small row group IDs with much larger ones. Merging partial blocks happens at different levels, and so does persisting them. However, in general, disorder in row group IDs can reduce coalescing efficiency.

Batch ingestion. The ordered row group IDs we rely on come from how data is ingested. We use a bulk ingest, which DuckDB uses its `PhysicalBatchInsert` operator (a `CREATE TABLE ... AS` or `INSERT INTO ... FROM` over a Parquet scan). If we enable `preserve_insertion_order`, DuckDB commits row groups in input order, so row groups get increasing IDs. Differential storage writes blocks in that same order, giving the near-contiguous layout of Figure 4.8. This is not guaranteed in general. With `preserve_insertion_order` disabled, DuckDB uses the `PhysicalInsert` path; row groups are written in an arbitrary, thread-dependent order that mixes the order and hurts coalescing. However, we still assume contiguous row group IDs because `preserve_insertion_order` is enabled by default in DuckDB and users tend to use that configuration. Furthermore, under `PhysicalInsert` the mixed IDs do not come from the writes but from the final

4.2 An analysis of differential storage layout on S3 - TPC-H, TPC-DS, and a MotherDuck workload

Benchmark	Column	Per-column			Single-buffer		
		Rg/file	Med	Max	Rg/file	Med	Max
TPC-H	<code>l_orderkey</code>	10	52	10126	5	24	291
TPC-H	<code>l_partkey</code>	64	154	443	9	73	315
TPC-H	<code>l_suppkey</code>	27	87	403	9	73	315
TPC-H	<code>l_extendedprice</code>	63	124	22678	9	78	374
TPC-H	<code>l_shipdate</code>	64	154	431	5	25	356
TPC-H	<code>l_comment</code>	31	180	567	14	136	418
TPC-DS	<code>ss_sold_date_sk</code>	46	79	129	3	27	104
TPC-DS	<code>ss_item_sk</code>	64	63	130	3	25	103
TPC-DS	<code>ss_ext_sales_price</code>	44	63	155	4	34	104
TPC-DS	<code>ss_ext_tax</code>	24	62	130	4	33	127
TPC-DS	<code>ss_net_paid</code>	24	62	191	4	32	147

Table 4.1: Row-group-ID distribution within S3 objects under the per-column and single-buffer writers (row-group size 122880, 64 threads). For each column, **Rg/file** is the median number of distinct row groups an object has for that column, and **Span** is $\max(\text{IDs}) - \min(\text{IDs})$ over those row groups; **Med** and **Max** are medium and max span, respectively. Per-column packs many row groups of a single column into each object (near-contiguous); single-buffer mixes all columns, so each object holds even fewer row groups.

per-thread merge, which is where row groups currently get their IDs. If instead we assign each row group a global increasing ID when it is flushed, and preserve that ID when merging the per-thread collections, the ID order would track the physical write order and stay increasing even under `PhysicalInsert`. We have not implemented this, but it is a small addition to the write path rather than a redesign.

In summary, consecutive row group IDs are grouped well: the median span is at most 180 on every column. The biggest spans in Table 4.1, 10,126 for `l_orderkey` and 22,678 for `l_extendedprice`, are not the typical case: each comes from a single outlier object. These are the objects involved in out-of-order flushes: because non-full blocks are held and merged near the end of the ingest, a single object can end up packing leftover blocks from row groups the writer ingested at different times. The pattern, consecutive row group IDs with some noise, complicates two strategies discussed earlier:

- **Caching.** Caching becomes useless, because row groups are consecutive and handed out consecutively: all threads read the same object at once and then move on together, so the cached object is consumed once and never revisited.

4. MOTIVATION AND DESIGN

- **Coalescing.** Coalescing nearby row groups together to create bigger requests would be less efficient if the noise dominates.

Multi-segment blocks As discussed in Chapter 2, DuckDB manages disk space efficiently using the `PartialBlockManager`, which tracks partially filled blocks and allocates the ones that still have enough space to host other segments. By doing this, DuckDB can pack data segments that do not cover a whole 256 KiB block. In principle, the 256 KiB block should be the main compression unit, so that every segment fits perfectly into this 256 KiB space. However, there are cases where a segment sinks below 256 KiB:

- **Compression.** A row group’s column data compresses below 256 KiB, so the whole column–row-group lives as one sub-block segment. This is the most likely case.
- **Residues.** A large column within a row group spans more than one block, so after filling several whole blocks its tail rows must form a segment that is not big enough to fill a final 256 KiB block.

We need to handle multi-segment blocks because they can reduce the benefit of coalescing in the `per_column` layout. A multi-segment block is hard to assign to a single per-column buffer, since it holds data from several columns; by default our design routes it to the column with the most bytes in the block, i.e. the dominant column. This creates noise in two ways:

- **Useless bytes.** Because the block now holds several segments, a query that needs only one of them still reads the whole block, fetching in the other segments, i.e., unnecessary bytes.
- **Coalescing.** A block’s minor segments are routed to the dominant column’s object, away from their own column’s object. If a query needs a minor column but not the dominant one, those segments cannot be coalesced with the rest of that column, so we must fetch each such 256 KiB block individually.

We first ask how prevalent they are, by classifying every 256 KiB block by the data segments it carries (Table 4.2): **single** (one data segment fills the block), **same column** (two or more segments, all from one (`table`, `column`), i.e. several row groups of one column packed together), and **mixed** (two or more segments from distinct columns). About 63% of blocks are single-segment, and 37% are multi-segment on both benchmarks. The same-column case is negligible (0.2% on TPC-H, effectively 0% on TPC-DS), so

4.2 An analysis of differential storage layout on S3 - TPC-H, TPC-DS, and a MotherDuck workload

Benchmark	Write mode	Blocks	Single	Same column	Mixed
TPC-H	<code>single_buffer</code>	531,855	63.3%	0.2%	36.6%
TPC-H	<code>per_column</code>	531,846	63.3%	0.2%	36.6%
TPC-DS	<code>single_buffer</code>	515,016	63.0%	0.0%	37.0%
TPC-DS	<code>per_column</code>	515,022	63.0%	0.0%	37.0%

Table 4.2: How data segments share a 256 KiB block, as a share of populated blocks, at row group size 122880 (SF=500). Each block is **single** (one data segment), **same column** (several row groups of one column), or **mixed** (distinct columns). Multi-segment blocks are almost always mixtures of different columns. This is the obstacle: blocks carry segments from more than one column complicate the effort to co-locate data blocks from the same column.

Benchmark	Segments	Compression	Residue
TPC-H	456,983	71.7%	28.3%
TPC-DS	491,858	58.5%	41.5%

Table 4.3: Why the segments packed into multi-segment blocks are below 256 KiB, per-column writer at SF=500. Each segment in a multi-segment block (≥ 2 data segments sharing one 256 KiB block) is classified by the total size of its containing column data (column, row group): **Compression** (parent ≤ 256 KiB, the segment is the whole column-row-group) or **Residue** (parent > 256 KiB, the segment is its partial tail). **Segments** is the total number of segments packed into such blocks.

essentially every multi-segment block mixes different columns. Write modes make no difference, as they operate on 256 KiB blocks at the file system layer.

Next, we ask which of the two causes puts the segments there. For every segment packed into a multi-segment block we look at the total size of its (column, row group): if that fits in ≤ 256 KiB the segment is the whole column-row-group that compressed below a block (**compression**); if the total size of this column-row-group exceeds a block, the segment is its leftover tail (**residue**). Table 4.3 gives the breakdown. Compression dominates on both benchmarks.

As we pointed out, there are two effects of multi-segment blocks; it is important to address them. For **coalescing**, the direct solution is pure per-column blob files. We can do this offline through compaction: instead of per-column blob files, we build per-column-combination blob files. For example, a block containing segments from columns A, B, and C is grouped into a blob that holds only blocks with that same A, B, C combination. This is only feasible offline, since the number of distinct combinations can blow up and

4. MOTIVATION AND DESIGN

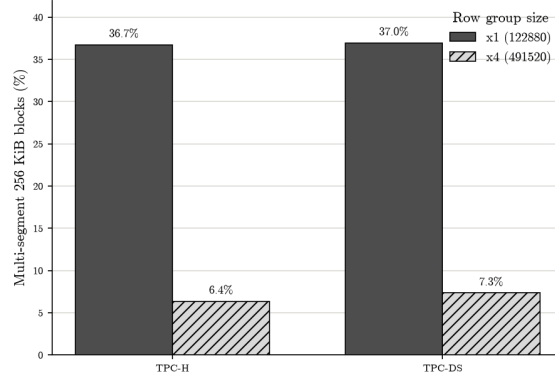


Figure 4.9: Increasing the row group size eliminates most multi-segment blocks. Share of 256 KiB blocks that are multi-segment (≥ 2 data segments), per-column writer at SF=500, for the default row-group size (x1 = 122,880 rows) versus x4 (= 491,520 rows).

would otherwise require too many in-memory buffers. This per-column-combination layout restores coalescing for such blocks, but the multi-segment blocks still survive, so the wasted-bytes problem remains. The second approach is to increase the row group sizes. A larger row group makes each column–row-group hold more data, so a greater portion of its segments fill whole blocks and fewer sub-block tails remain to be packed together. Figure 4.9 confirms this: quadrupling the row group from the default 122,880 rows to 491,520 (x4) cuts the share of multi-segment blocks from 37% to 6.4% on TPC-H and 7.3% on TPC-DS, roughly a six-fold reduction, effectively eliminating most multi-segment blocks.

Implications for the design. The layout analysis of TPC-H and TPC-DS at SF=500 gives a few key findings:

- **S3 objects mix many columns.** Under the default single-buffer writer, an object hosts a median of 16 distinct columns on TPC-H and 24 on TPC-DS, and almost none keeps only a single column. This prevents coalescing and makes optimistic caching wasteful. The per-column writer keeps data from the same column in the same objects, enabling efficient prefetching.
- **Row group order is near-sequential but noisy.** Within an object, row group IDs are mostly contiguous, but parallel writing adds noise. Consecutive row groups dispatched consecutively reduce cache efficiency, and outliers lower the chance of coalescing.

4.2 An analysis of differential storage layout on S3 - TPC-H, TPC-DS, and a MotherDuck workload

- **Multi-segment blocks mix columns.** About 37% of 256 KiB blocks contain segments from two or more columns, on both TPC-H and TPC-DS. Compression is the main cause. Grouping multi-segment blocks into the same per-column-combination files keeps coalescing efficient even with these blocks; however, reading wasted bytes remains a problem.
- **A larger row group is an effective configuration.** Increasing the row group size (x4) reduces multi-segment blocks about six-fold, from 37% to 6.4% on TPC-H and 7.3% on TPC-DS, and gives a cleaner, more coalescible layout.

4.2.3 A real-world workload at MotherDuck

TPC-H and TPC-DS are bulk-loaded in a single ingest, which is a friendly scenario for the write path. To validate whether the findings of the previous sections carry over to production, we apply the same block-level analysis to MotherDuck’s internal data warehouse (MDW), which contains a large volume of operational logging data, e.g., API logs and usage data. This live DuckDB database has been ingested and compacted over a long period, and we also replay its query history to characterize the read side. MDW differs from the benchmarks in two important ways: ingestion is incremental rather than one-shot, and the workload is dominated by wide tables with significant text columns.

How MDW is ingested. Data comes from different sources, but in general, all raw data first lands in S3; logging data is the largest source. A pipeline combines them into landing tables, after which the data is transformed into consumable DuckDB tables, from which MDW serves queries. The write patterns to these tables vary. Some tables with daily usage are fully rebuilt via `CREATE OR REPLACE TABLE AS`. Others are maintained using a delete-insert strategy: newly arriving rows likely already exist in the table, so they are deleted and reinserted. A third group is append-only, and some tables bypass the transformation pipeline entirely, being appended directly every 15 minutes.

Block-level layout. We redefined the categories of Table 4.2 into five classes: **A** (a single segment), **B** (multiple segments, one column and one row group), **C** (one column, several row groups), **D1** (multiple columns, one row group) and **D2** (multiple columns, several row groups). B and C are same-column packing, while D1 and D2 are the mixed blocks that cause wasted reads and broken coalescing under a per-column layout. The breakdown is shown in Figure 4.11.

4. MOTIVATION AND DESIGN

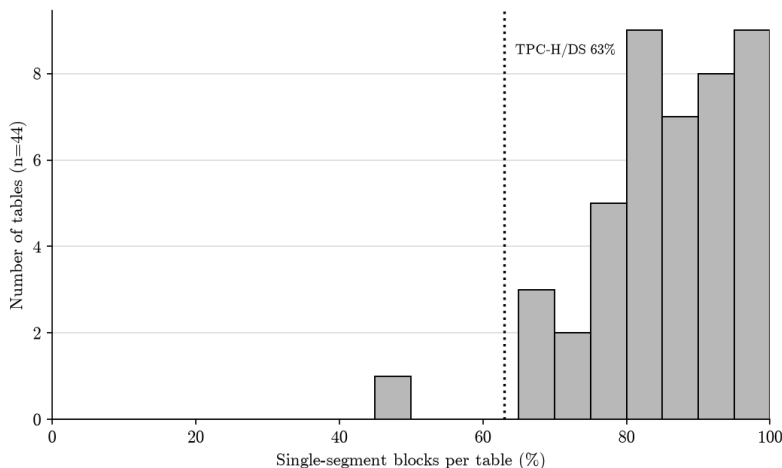


Figure 4.10: Distribution of the single-segment block share across MDW warehouse tables with at least 1,000 blocks. Production tables have more single-segment blocks (about 85% on average) than TPC-H/TPC-DS (63% on average), but a tail of tables remains high.

On MDW, the single-segment share is higher than on TPC-H and TPC-DS. Across tables with at least 1,000 blocks, on average about 85% of blocks are single-segment (Figure 4.10), compared with the 63% we measured on TPC-H and TPC-DS. For some large tables, storage is dominated by a single wide text column whose values fill entire 256 KiB blocks on their own (every overflow block we observe is a continuation of such a text column), which accounts for the majority of single-segment counts. Setting those dominant text columns aside, the share decreases, and some tables drop below 70%, some near 50% of blocks. The remaining multi-segment blocks are almost entirely **D1** blocks that pack many small segments of a single row group. Same-column packing (B and C) is negligible, as on TPC-H and TPC-DS, and essentially no block mixes different tables.

How MDW is queried. We replay the production query log. For each query, we read its SQL to find which columns of each table it touches, and assume it scans those columns in full. We then compare the bytes the query needs against the bytes it would read if it fetched whole 256 KiB blocks.

Two things stand out. First, queries read few columns: about 75% of (query, table) pairs touch at most 30% of a table’s columns (Figure 4.12). Most of a table’s bytes are useless to most queries, so co-locating each column and reading only the needed ones in large requests is a clear win. Second, reading whole blocks wastes bytes. The unneeded share of the bytes a query reads is 15% at the median and 82% at the 95th percentile (Figure 4.13a). Overall, this is only about 1.6% of the bytes read because the large text columns sit in

4.2 An analysis of differential storage layout on S3 - TPC-H, TPC-DS, and a MotherDuck workload

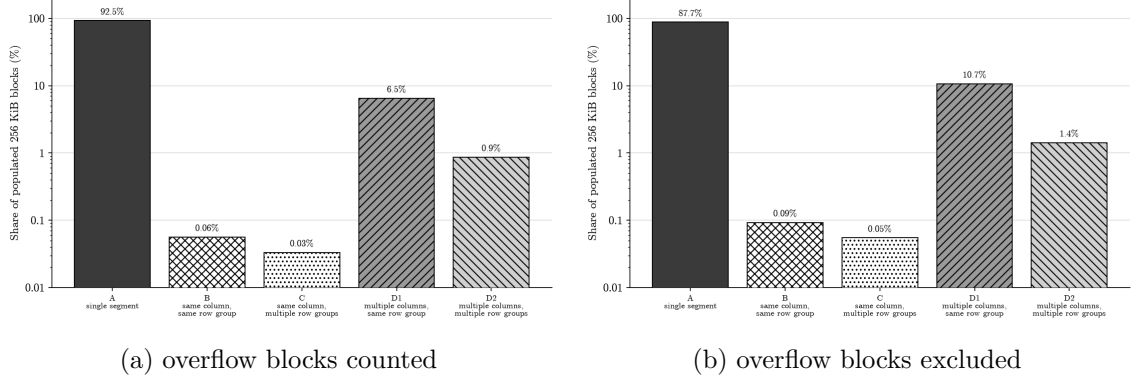


Figure 4.11: Block category breakdown over MDW warehouse blocks (log scale). (a) counts the overflow blocks; (b) excludes them. Multi-segment blocks are mostly **D1** (multiple columns, one row group), the blocks that pack small columns together; excluding overflow raises the multi-segment share from about 7.5% to about 12%. Same-column packing (B, C) is negligible in both.

their own blocks and waste nothing. The waste falls on the small, frequently read columns: queries that need at most 1 MiB read about $3.7\times$ that in unneeded data (Figure 4.13), because their small segments are packed into multi-segment (D1) blocks, and one such query can touch over a hundred thousand of them.

Implications for the design. The production workload suits our approach even better than the benchmarks:

- **Per-column layout fits.** Blocks hold one table (essentially 100%), most blocks hold a single segment (about 85%), and queries read few columns (75% touch $\leq 30\%$). All of this favors per-column objects and large, coalesced reads.
- **Multi-segment blocks still hurt.** The median query wastes about 15% of the bytes it reads, and small queries far more (Figure 4.13b). Removing D1 blocks, by compacting into per-column-combination objects (Section 4.2.2) or using larger row groups, recovers those reads.

4.2.4 Summary

In this section, we studied how data is organized on cloud object stores with differential storage and pointed out important strategies to make better requests, i.e., larger requests:

- **The single-buffer writer mixes data within S3 objects.** Maintaining a single buffer makes it hard to achieve larger request sizes. This suggests maintaining one

4. MOTIVATION AND DESIGN

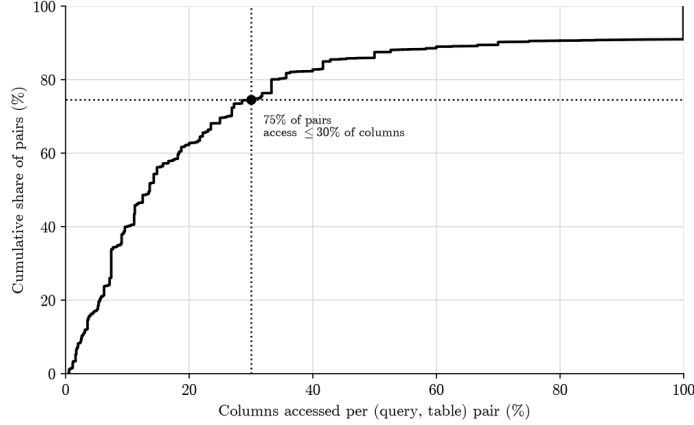


Figure 4.12: Column selectivity on MDW: cumulative distribution of the fraction of a table’s columns accessed per (query, table) pair. About 75% of queries touch at most 30% of columns.

buffer per column, so that data from the same column is co-located in the same S3 objects.

- **Row group order:** row groups tend to stay close together within a blob, which is beneficial for coalescing. However, it could lead to inefficient caching if we distribute row groups to workers in sequential order.
- **Multi-segment blocks:** multi-segment blocks could potentially reduce the effectiveness of coalescing. This could be mitigated with compaction and larger row groups.

In this section, we have developed an intuition for making S3 requests bigger via differential storage. However, the concurrency problem remains and depends on the number of threads a DuckDB instance has. We address this problem by implementing a prototype prefetching mechanism that fetches row groups before passing them to worker threads.

4.3 Write path

In this section, we describe how we implement the write path for our experiments. As MotherDuck’s differential storage system has a more complex architecture to provide additional features to customers, we decided to implement a simplified version of differential storage via the DuckDB extension mechanism, which allows us to plug in our own file system.

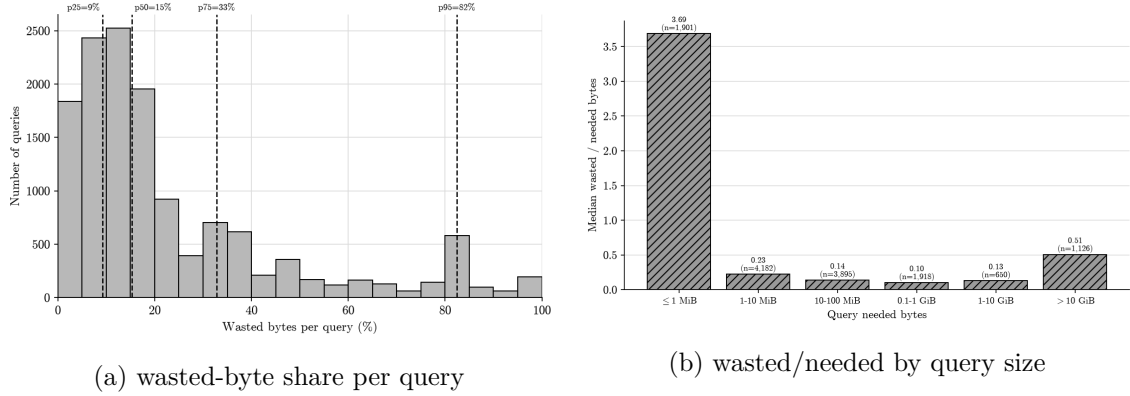


Figure 4.13: Wasted reads from reading whole blocks. (a) Per-query share of fetched bytes the query does not need (median 15%, tail past 80%). (b) Wasted bytes relative to the bytes a query needs, bucketed by needed-byte size: queries needing at most 1 MiB read a median of $3.7\times$ their needed bytes in unwanted data, because they hit many small segments packed into multi-segment blocks, while large scans waste little.

4.3.1 A simple S3 file system

The DuckDB extension lets us plug in our own file system, so we implement a file system called `SimpleS3FileSystem`. The two important interfaces DuckDB uses to interact with the file system are shown in Listing 4.1.

Listing 4.1: Read and Write methods

```
DUCKDB_API virtual void Read(FileHandle &handle, void *buffer,
int64_t nr_bytes, idx_t location);

DUCKDB_API virtual void Write(FileHandle &handle, void *buffer,
int64_t nr_bytes, idx_t location);
```

DuckDB reads and writes to a single local file using the two calls `Read` and `Write` (Listing 4.1), each with a logical byte position and a buffer. A `Write` is not applied to a physical file: we append the write (typically a block) to an in-memory buffer that will later become an S3 object, and we keep track of where that logical position lives on S3. A `Read` looks the logical range up and fetches the block from its object. `SimpleS3FileSystem` is the intermediate layer that does the translation, making it transparent to DuckDB. The mapping lives in the mapping manager (`SimplePersistedS3MappingManager`), a flat array indexed by logical block: entry i describes the i -th 256 KiB block of DuckDB’s file or three header blocks. Each entry is an `S3FileInfo` (Figure 4.14): the S3 object key, the offset of the block inside that object, the write type, and the list of segments the block

4. MOTIVATION AND DESIGN

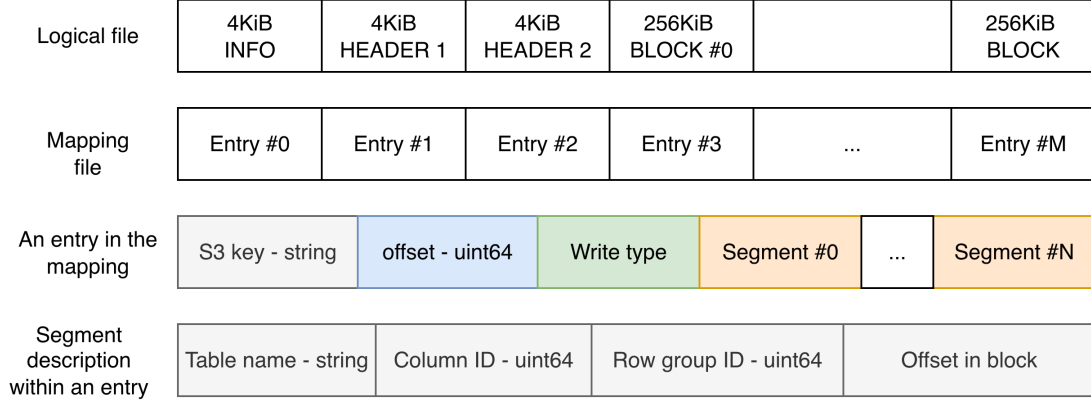


Figure 4.14: How a logical file is tracked via the flat-array-style mapping file. The three 4 KiB header blocks correspond to the first three entries in the mapping; after that, each data block maps to one entry in the mapping, following the order in the logical file. Each entry contains the necessary information to locate the blocks on S3, including an S3 object key, an offset, a write type (metadata or data), and additional context information for each segment within the 256 KiB block. Each segment description includes information about the table, column, and row group it belongs to.

contains.

The mapping is persisted to the local file system when DuckDB calls `fsync` on the logical file. `SimplePersistedS3MappingManager::Sync` serializes the array and writes it out as one small mapping file (Listing 4.2) on the local disk. This is not an in-place update: each `Sync` deletes the old mapping file and rewrites the entire mapping. This is a deliberate simplification: it is cheap because the mapping is small compared to the data it indexes, but it rewrites every entry each `fsync`. On open, we read the file back and rebuild the array in memory. The design is not memory-efficient, as values are neither compressed nor organized efficiently. However, as it is simple, accessing a single block is an $O(1)$ lookup. Considering the memory footprint, describing TPC-H and TPC-DS requires a mapping file with sizes ranging from 40 MiB to 60 MiB (90 bytes to 135 bytes per entry). A rough estimate is that if a database file is 1 TiB, i.e., 4 million blocks, it would require 4 million entries to describe the file, which means the mapping is around 400 MiB with 100-byte entries. This is acceptable for our case, as we are working with an instance having large memory. Adding the context, i.e., segment information, incurs 43% more space for the mapping on TPC-H and 63% on TPC-DS. Most of the memory is used to store the strings for S3 keys and the table names, which can be optimized by using compression or a different convention. For example, using integers to describe table names and S3 keys.

Listing 4.2: Every `SimplePersistedS3MappingManager::Sync` replaces the whole mapping file: the old file is deleted, then all entries are written back. Nothing is updated in place.

```
def sync():
    mapping = serialize(entries) # DuckDB fsync'd the logical file
    delete(mapping_path)        # the entire mapping, not a delta
                                # the old file goes first ...
    write(mapping_path, blob)    # the mapping is written back
```

4.3.2 Single buffer

This is similar to the current writer at MotherDuck, in that it uses no context. There is one active buffer; every block DuckDB flushes is appended to it in the order of flushing. When the buffer reaches 16 MiB in size, it is persisted to one S3 object, the mapping entries for its blocks are populated, and a fresh buffer starts. It is simple, but an object then holds whatever happened to be flushed together, usually a mix of many columns (Figure 4.5), which is poor for coalescing and caching.

4.3.3 Per-column buffer

To keep a column's blocks together, the differential storage must know what each block contains, which DuckDB does not tell the file system. Therefore, we need to instrument DuckDB to pass this information. During ingestion, the `PartialBlockManager` packs many column segments into each 256 KiB block, and for each block it records a `PartialColumnSegment`: the column, the row group, and the offset within the block. We patch the write path so that this list is not thrown away. When a partial block is written out, `CreateFlushContext` wraps the segment list with a write-type tag (data, metadata, index, or string overflow), in a `StorageWriteContext` and puts it into the `QueryContext` passed to the file system. (Listing 4.3).

Listing 4.3: DuckDB patch: attach the block's segments to the flush context (pseudocode)

```
def create_flush_context(block, free_space):
    if block holds column data:
        # the only case that carries the segment list
        ctx = StorageWriteContext(DATA, block.segments, free_space)
    else:
        ctx = StorageWriteContext(UNSORTED)
    return QueryContext(ctx) # handed to the file system
```

4. MOTIVATION AND DESIGN

Our `SimpleS3FileSystem` then reads this context during a write, extracting each `PartialColumnSegment` into a `SegmentInfo` (table, column id, row group, offset in the block), and passes it to the buffer manager.

In per-column mode, the file system maintains one active buffer per column. For each block, it picks the **dominant column**, the (table, column) that has the most bytes in the block, and appends the block to that column’s buffer. Each buffer fills its own objects, so almost every object is dominated by a single column. A downside of this per-column writer is that we maintain multiple in-memory buffers. For wide tables with many attributes, this can become costly; for example, a table with 100 columns would require 1.6 GiB of memory for the buffers alone.

The same segment list is also stored in the block’s mapping entry (`S3FileInfo.segments`), so the mapping tracks not only where a block lives but what it holds. This allows the compaction pass to reconstruct the (table, column, row group) for every block. Storing this additional information, of course, incurs extra overheads as discussed in Section 4.3.1, typically adding around 40%–60% (20 MiB–44 MiB) to the memory for TPC-H and TPC-DS with a scale factor of 500.

4.3.4 Compaction

We can not split a block further, so per-column mode cannot separate the minor columns that stay in a multi-segment block; they stay in the dominant column’s objects (Section 4.2.2). We remove this with a compaction pass (Algorithm 1), implemented as an offline tool that reads the stored segment lists and rewrites the objects so that blocks are grouped by the full set of columns they contain (a per-column-combination layout), with the blocks inside each object ordered by row group. Grouping by combination places every multi-segment block with the same column set in the same objects, and ordering them by row group turns each object into a contiguous run. Hence, a scan of those columns coalesces again. This has to run offline because doing it while writing would require one open buffer per combination, and the number of combinations can grow unbounded. Compaction thus trades a background rewrite for cleaner, more coalescible objects.

4.4 Read path

We now move on to how we manage the read path of our file system.

Algorithm 1 Offline compaction into grouped objects

```

1: function COMPACT(mapping)
2:   clusters  $\leftarrow \emptyset$  ▷ key  $\rightarrow$  list of blocks
3:   for all block b  $\in$  mapping with a non-empty key do
4:     if b.write_type  $\neq$  DATA then
5:       key  $\leftarrow$  META ▷ metadata, index and header blocks share one buffer
6:     else
7:       key  $\leftarrow$  set of distinct (table, column) pairs in b.segments
8:     end if
9:     append b to clusters[key]
10:  end for
11:  for all (key, blocks)  $\in$  clusters do ▷ same key packed together
12:    sort blocks by (MINROWGROUP(b), original index) ▷ ordered runs
13:    buffer  $\leftarrow$  new 16 MiB object
14:    for all b  $\in$  blocks do
15:      if buffer is full then
16:        PERSIST(buffer); buffer  $\leftarrow$  new 16 MiB object
17:      end if
18:      off  $\leftarrow$  append b's bytes to buffer
19:      new_mapping[b].key  $\leftarrow$  buffer.key; new_mapping[b].offset  $\leftarrow$  off
20:    end for
21:    PERSIST(buffer)
22:  end for
23:  write new_mapping ▷ the input mapping and its objects are left intact
24: end function

```

4.4.1 Synchronous read

On a read request from DuckDB, we look up the block's destination in the mapping manager and issue the corresponding request to S3. DuckDB implements synchronous I/O, so for every read we issue the request to S3 and wait for the data to arrive before returning it (`StandardCacheManager::Read`). How the request is issued is set by a read mode (`ReadMode`), described next.

4.4.1.1 Without caching

This is a naive approach: we issue requests on demand, only when DuckDB needs them, i.e. fetching individual 256 KiB blocks from S3 (`ReadMode::DIRECT`). A 256 KiB S3 request

4. MOTIVATION AND DESIGN

is slow, with a latency of 30–70 ms. For example, in an ideal scenario where every 256 KiB request costs 50 ms and the DuckDB engine has 64 threads, we can only sustain a maximum throughput of about 320 MiB/s ($= 64 \times 256 \text{ KiB} \times 1000/50$), orders of magnitude lower than the network bandwidth.

In this mode, the direct read path does one thing to improve that: *physical coalescing*. DuckDB’s own prefetching logic groups the blocks of a row group and merges those with consecutive IDs into large ranges before handing them to the file system; when our extension receives such a range, if its 256 KiB blocks map to contiguous offsets within the same S3 object, we merge them into a single larger GET. How often this fires is decided entirely by the logical write layout, since only logically consecutive blocks that the writer also placed physically next to each other can be merged.

4.4.1.2 With caching

An obvious first improvement is to read the entire 16 MiB object when any of its blocks are requested. The mechanism is simple: any 256 KiB block read also triggers a 16 MiB fetch to a local disk cache. This is easy to implement, and in our file system we manage this cache with an LRU queue of 16 MiB entries. Also, at MotherDuck, a caching layer already sits at several points along the read path, making this easy to enable. We can implement this in two ways (`ReadMode`):

- **Synchronous read** (`ReadMode::SYNC_CACHE`): on every read, we issue a read for the entire 16 MiB object, wait for it to arrive, and serve 256 KiB back to DuckDB.
- **Asynchronous read** (`ReadMode::ASYNC_CACHE`): upon every read, we issue the read for the 256 KiB block, and in the background trigger the 16 MiB fetch into the local disk cache.

With this optimistic read approach, a 256 KiB block read pulls a full 16 MiB into cache. Given all cache hits and the ideal scenario in which consecutive 256 KiB reads land in different objects, we hide the 16 MiB fetch latency behind the 256 KiB latency. We can, in principle, sustain about 20 GiB/s (64 threads, 50 ms latency, 16 MiB per fetch). However, this theoretical throughput is often significantly inflated because many 256 KiB reads end up in the same object, or because DuckDB’s processing speed outruns the 16 MiB fetch latency.

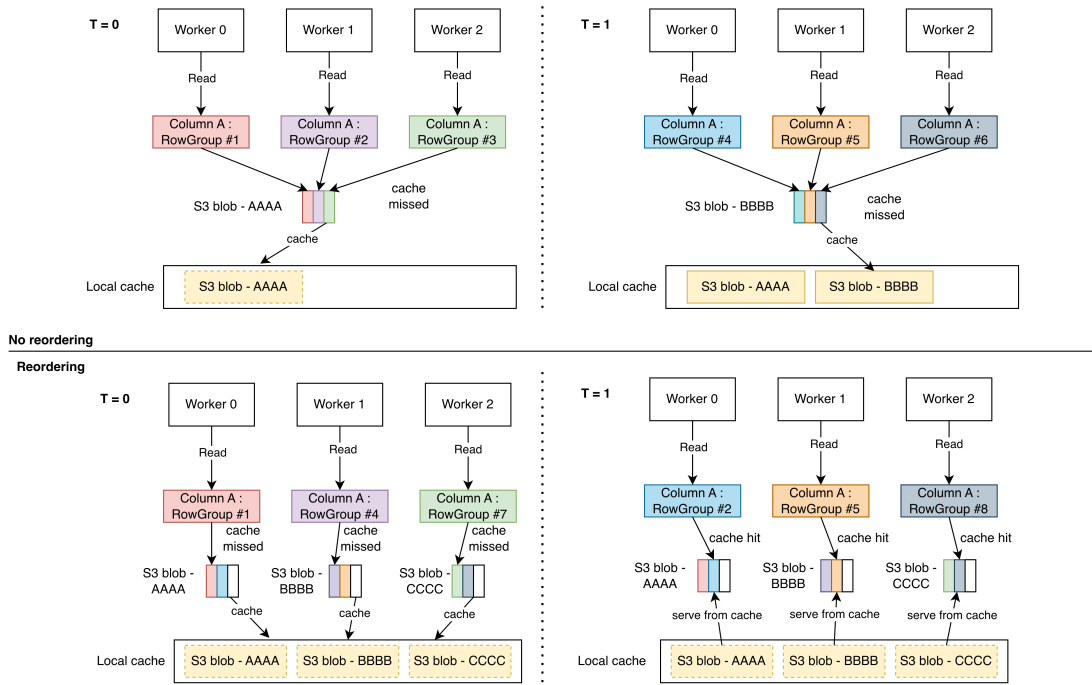


Figure 4.15: Why caching alone fails. An example of inefficient caching caused by sequential row group dispatching and contiguity of row group IDs on S3 objects. A stride order then helps to increase the hit rate.

4.4.2 Reorderer

Figure 4.15 compares the two dispatching strategies over two consecutive time steps, with three workers scanning **Column A**. In the **no reordering** case (top), DuckDB gives out row groups sequentially, so at $T = 0$ all three workers receive row groups 1–3. Because the per-column layout packs consecutive row groups together, these all reside in the same S3 object (AAAA); the workers therefore miss the cache simultaneously and race to fetch the same blob. We then cache this AAAA S3 object. At $T = 1$ they move to the next batch of row groups 4–6, which likely stay in a single new object (BBBB), leading to another round of simultaneous misses. The cached copy of AAAA is never reused, so the hit rate stays near zero. The **reordering** case (bottom) instead spreads each worker across a different region of the column: at $T = 0$ the workers read row groups 1, 4, and 7, which map to three distinct objects (AAAA, BBBB, CCCC) that are all cached. At $T = 1$ the workers move to row groups 2, 5, and 8, each of which lives in an object already stays in the local cache, turning every access into a cache hit.

DuckDB has a built-in optimizer called **RowGroupPruner**, which sorts row groups to

4. MOTIVATION AND DESIGN

enable quick exits in some queries (for example, `LIMIT` and `TOP N`). Row groups are then no longer handed out sequentially, but by their zonemap values (`RowGroupPruner::CreateRowGroupReordererOptions`). `RowGroupPruner` installs a reorderer during scanning: the scan consults whether it has a reorderer, and if so asks the reorderer to hand out row groups in its own order; otherwise it distributes row groups in sequential order (`scan_state.cpp`, Algorithm 2). We find this logic easy to extend, so we patch DuckDB in two small ways: the reorderer now takes a custom ordering function (`custom_sort` on `RowGroupOrderOptions`), and any optimizer can install one on a `LogicalGet` through `SetScanOrder` (backed by the table function’s `set_scan_order` hook). We then add an optimizer, `RowGroupOrderOptimizer`, that installs a reorderer distributing row groups in a modulo fashion: it buckets them by `row_group_id % modulo` (default 64) and serves one bucket at a time, so consecutive row groups, which sit close together on S3, are read at different times.

When a thread fetches the next row group (Algorithm 2), the scan state asks for the next row group and first checks whether a reorderer was installed (from the plan’s `SetScanOrder`). If one is present, it delegates to `reorderer->GetNextRowGroup`; otherwise it falls back to the segment tree’s sequential order (`GetNextSegment`). At the reorderer, the order is materialized once: on the first row group `GetRootSegment` builds the full ordered list, running our `custom_sort` to bucket the row groups, and each later `GetNextRowGroup` returns the next entry in that list, advancing an internal offset and returning null when it is done. So the modulo order is decided up front.

4.4.3 Prefetching

In this section, we discuss how we do efficient prefetching with the reorderer. The reorderer provides a convenient hook for prefetching. The idea is to prefetch data before we hand out row groups to worker threads. This point is convenient for the following reasons:

- We have complete control over what to prefetch: the scan state tells us what the scan will touch, so during the scan we know which row groups are needed, which column data is needed, and which will be skipped due to filtering. As a result, we can prefetch exactly that data into memory.
- We also have a complete picture of the data layout on S3 from the mapping, enabling us to coalesce requests.

Algorithm 2 Handing out the next row group with a reorderer

```

Scan level                                ▷ falls back to sequential order if no reorderer is installed
1: function GETNEXTROWGROUP(current)
2:   if scan has a reorderer then
3:     return reorderer.GETNEXTROWGROUP(current)
4:   else
5:     return SEQUENTIALNEXT(current)                                ▷ segment-tree order
6:   end if
7: end function

Reorderer                                ▷ order materialized once, then walked
8: function GETROOTSEGMENT(row_groups)
9:   ordered  $\leftarrow$  CUSTOMSORT(row_groups); offset  $\leftarrow$  0
10:  return ordered[0]
11: end function

12: function GETNEXTROWGROUP(current)
13:   if offset  $\geq$  |ordered| - 1 then
14:     return NULL                                                    ▷ scan finished
15:   end if
16:   offset  $\leftarrow$  offset + 1
17:   return ordered[offset]
18: end function

19: function CUSTOMSORT(row_groups)    ▷ our modulo ordering, default modulo = 64
20:   return Sorted list of row_groups by (id mod modulo, id)
21: end function

```

With prefetching, we could prefetch aggressively and precisely to achieve optimal throughput by scaling the number of concurrent requests to S3 up and down.

The high-level idea is as follows. We maintain a window of row groups to prefetch and a queue of row groups that are being prefetched. When a thread requests a row group, we check the top of the queue: if the data for that row group has arrived, we hand it to the thread. At the same time, we issue more prefetch requests as the window drains. Algorithm 3 shows the pseudo-code for the reorderer-based prefetcher.

4. MOTIVATION AND DESIGN

Algorithm 3 Reorderer-based prefetcher

```
1: function GETROOTSEGMENT(row_groups)
2:   row_group_list  $\leftarrow$  row_groups
3:   REFILL ▷ start prefetching the first window row groups
4:   return WAITANDPOP
5: end function

6: function GETNEXTROWGROUP ▷ a thread finished a row group, wants the next
7:   REFILL ▷ top up prefetches as the window drains
8:   return WAITANDPOP
9: end function

10: function REFILL
11:   while in_flight < window and row_group_list not exhausted do
12:     rg  $\leftarrow$  next row group in row_group_list
13:     blocks  $\leftarrow$  COLLECTBLOCKS(rg)
14:     ISSUE(blocks) ▷ async S3 fetches
15:     ENQUEUE(rg)
16:   end while
17: end function

18: function WAITANDPOP
19:   rg  $\leftarrow$  front of queue; wait until all of its blocks have arrived
20:   return rg ▷ hand back the ready row group
21: end function
```

4.4.3.1 Without batching

In this scheme, we collect the data blocks needed during the scan and issue individual 256 KiB requests. We maintain a thread pool of 1024–2048 threads to handle the individual requests, a number taken from the baseline benchmark in the previous section. All of these requests are asynchronous.

4.4.3.2 With batching

In this scheme, instead of simply collecting single blocks and issuing them separately, we coalesce the requests on S3. Since we have full information about where each block lives on S3, we take a batch of consecutive row groups (around 256–1024), collect their blocks,

sort them by position within the S3 objects, and merge consecutive positions into a single request. This is based on the fact that row groups IDs likely stay consecutively on object files Section 4.2.4. As a result, instead of many small and separate S3 requests, we issue fewer, larger ones, exploiting the per-column layout we built on the write path. The merged request is issued asynchronously through the AWS S3 client [10].

Request hedging Batching introduces a tail latency: a single slow request can stop the row group at the front of the queue being dispatched and block all worker threads, even when most of its blocks have already arrived. To prevent this, we add a threshold on how many blocks the top row group may be missing when a thread requests that row group. If it is missing at most that many blocks, we still consider it finished, hand it out, and delegate the missing blocks to individual 256 KiB requests, which are reissued on demand when the scan actually reads them. This trades a few small extra requests for not stalling the scan on a straggling batch. With the threshold set to zero, the hedge is disabled, and the consumer waits for every block to arrive.

4.5 Memory considerations

Several of our designs keep data in memory beyond what DuckDB itself needs, on both the write and the read path. Throughout this thesis, we assume the compute instance has enough memory for this, which is reasonable for the analytics instances we target (the `m6idn.16xlarge` used in our benchmarks offers 256 GiB), since these buffers are a small fraction of the available memory.

On the **write path**, the per-column writer (Section 4.3.3) keeps one 16 MiB active buffer per column. For a wide table, this adds up: the widest TPC-DS tables, at 34 columns, need about 0.5 GiB each if all columns are ingested at once, and roughly twice that when each nullable column’s validity mask is buffered alongside it. On the **read path**, the prefetcher (Section 4.4.3) fetches aggressively ahead of the scan: it holds a window of row groups, together with their in-flight requests, in memory until a worker thread consumes them. This uses memory to hide latency, but the window size and the number of in-flight requests bound the memory usage. Both of them are configurable, allowing it to be tuned to the available memory.

A more memory-conscious design, for example one that caps the number of active buffers or adapts the prefetch window and falls back under memory pressure, is left as future work.

4.6 Putting it all together

In this section, we summarize all the experimental designs we have discussed so far and describe our hypotheses. Each design pairs a write layout (single buffer or per-column, optionally with larger row groups and compaction) with a read strategy (synchronous, with or without caching, or reorderer-based prefetching, with or without batching). Broadly, we expect performance to climb as we move from on-demand synchronous reads, to caching, and then to prefetching with coalescing on a clean per-column layout.

Synchronous read without caching, on the single-buffer and per-column layouts

This is the current mechanism, and our baseline. Every 256 KiB block is fetched on demand, and the thread waits for it, so throughput is bounded by the per-request latency multiplied by the number of threads, about 320 MiB/s at 64 threads and 50 ms per request. The write layout does not matter here because we never fetch more than the one block requested, so there is nothing to coalesce or reuse. We expect this to have the lowest performance of all designs.

Synchronous read with asynchronous or synchronous caching, on the single-buffer layout

Here each block read also pulls the whole 16 MiB object into cache. In the single-buffer layout, an object mixes many columns (a median of 16 on TPC-H and 24 on TPC-DS), so most of the 16 MiB we fetch is data the query does not need. Those extra bytes rarely become future hits, so the hit rate remains low, and the larger fetches mostly add unnecessary overhead. We expect little performance improvement over the no-cache baseline, and possibly worse if the wasted bandwidth dominates.

Synchronous read with asynchronous or synchronous caching, on the per-column layout and on compacted layout

On the per-column layout, a single column dominates an object. Hence, a 16 MiB fetch is almost entirely useful to a scan of that column, and caching should help. However, DuckDB still hands out row groups sequentially, and consecutive row groups sit close together on S3, so all threads read the same object at once and then move on together; the cached remainder is consumed immediately rather than reused later. Because of this sequential row group problem, we expect caching to help but not by a big jump.

Synchronous read with caching, on the per-column layout or compacted layout and with the modulo reorderer enabled The reorderer breaks the sequential hand-out: it buckets row groups by their ID modulo a constant. It dispatches one bucket at a time, so the threads work on row groups that live in different objects, and an object fetched now is reused a few steps later. We expect that caching will be beneficial here because the cached objects are both useful (per-column) and reused (via strided access).

Reorderer-based prefetching without batching, on either the single-buffer or per-column layout Instead of a thread waiting on each block, a pool of 1024–2048 threads issues the reads ahead of the scan. Without batching, we issue individual 256 KiB requests, so the layout does not matter. Still, the prefetch hides the per-request latency. We expect this to be better than synchronous read, and to reach roughly the throughput seen in the baseline section, around 4 GiB/s.

Reorderer-based prefetching with batching, on the single-buffer layout With batching, we coalesce nearby blocks into fewer, larger requests. On the single-buffer layout, though, a column’s blocks are scattered across objects and interleaved with other columns, so there is little that is contiguous to merge. We therefore expect little performance gain over prefetching without batching.

Reorderer-based prefetching with batching, on the per-column layout On the per-column layout, a column’s blocks are contiguous within an object, so batching can merge them into large requests and cut the request count sharply. We expect a performance improvement here. The ideal 16 MiB request size still cannot be reached, because multi-segment blocks leave small foreign segments interleaved in a column’s objects and break some of the contiguous runs.

Reorderer-based prefetching with batching, on the per-column layout with increased row group sizes and a compacted layout Increasing the row group size and compacting the layout removes most multi-segment blocks. Hence, a column’s objects become almost purely that column. The number of requests should drop significantly while throughput stays comparable or improves. We expect this to be the best option: the fewest and largest requests on the cleanest layout.

4. MOTIVATION AND DESIGN

5

Performance evaluation

In this chapter, we set up our experiments to evaluate the different designs discussed in Chapter 4 and the hypotheses we proposed.

5.1 Evaluation setup

5.1.1 Benchmark suites

We use the two widely used decision-support benchmarks, TPC-H and TPC-DS, to evaluate our configurations. Both are proposed by the Transaction Processing Performance Council (TPC), a non-profit entity setting industry-standard database benchmarks. TPC-H and TPC-DS are OLAP benchmarks that involve complex queries that scan large volumes of data. Both benchmarks additionally define data-maintenance work: TPC-H has two refresh functions that insert and delete `orders` and `lineitem` rows, and TPC-DS has a maintenance phase that applies inserts and deletes to its fact tables. We run neither: this evaluation targets the read path, and every dataset here is written exactly once by a bulk load, so each layout we measure is the layout such a load produces.

TPC-H is the older and simpler of the two. It models a wholesale supplier business with just 8 tables in a normalized schema. Its workload comprises 22 read-only queries that involve joins, aggregations, subqueries, and sorting. TPC-H remains an often-used benchmark for analytical databases. However, it is often criticized for not being representative of a real-world workload.

TPC-DS, first released in 2006, was designed to address those criticisms. It models a retail company using a schema with 24 tables (7 fact tables and 17 dimension tables). The workload is more demanding: 99 query templates covering reporting, ad-hoc analysis, and data-mining-style queries.

5. PERFORMANCE EVALUATION

Component	Specification
Instance type	<code>m6idn.16xlarge</code>
vCPUs	64
Memory	256 GiB
Local storage	2 × 1900 GB NVMe SSD (2.5 GiB/s at 256 KiB I/O)
Network bandwidth	100 Gbps (11.6 GiB/s)
DuckDB	v1.5.0-dev with our patches

Table 5.1: Compute and storage configuration used in our experiments.

We use DuckDB’s built-in TPC-H and TPC-DS extensions first to generate the tables as Parquet files, using DuckDB’s default configuration. We then ingest these Parquet files back into DuckDB through our extension, so that the data ends up stored on S3. The ingest also produces the mapping files, which reside on the local file system of the DuckDB instance. The only knob we change during ingestion is the row group size, which we state explicitly when it matters.

5.1.2 Compute and storage infrastructure

We run all experiments on a single `m6idn.16xlarge` instance, a balanced configuration of compute, memory, network bandwidth, and local disk; its specifications are listed in Table 5.1. The network bandwidth (11.6 GiB/s) exceeds the throughput of a single local NVMe SSD (2.5 GiB/s). Surprisingly, the development of cloud-based SSDs is not keeping up with that on commodity laptops or with network bandwidth, which can now achieve 15 GiB/s. Cloud-based SSDs are currently capped at around 2 GiB/s per SSD [32]. One solution is to use multiple SSDs and an orchestrating layer to distribute I/O blocks across disks, thereby achieving the cumulative throughput of several SSDs. However, this complicates the design, so we opt for a single SSD.

5.1.3 Notation

Throughout the evaluation, we compare three write layouts, each the output of a different writer for the same data. **Single-buffer** is differential storage’s naive writer: a single active buffer shared across all threads, to which every flushed 256 KiB block is appended in flush order and persisted 64 blocks (16 MiB) at a time. **Per-column** is a writer that instead routes each block to a per-column buffer, co-locating a column’s blocks in the same S3 objects. **Compacted** is the offline per-column-combination layout produced by the

compaction pass (Chapter 4), which re-groups the remaining multi-segment blocks by the full set of columns they contain. We also vary the row group size: **rgsx1** (abbreviated **x1**) is DuckDB’s default of 122,880 rows, and **rgsx4** (**x4**) is four times that (491,520 rows).

5.2 Performance criteria

We evaluate all benchmark queries in a *cold* setup to eliminate the effects of inter-query caching: after each query, we wipe the local object cache and re-attach the database on a fresh connection (additionally drop the OS page cache), so that every query is measured independently and reflects cold-access latency. In addition, because S3 performance is unstable [27], we run the benchmark several times over different days and report the average results.

Our primary performance indicator is query execution time. Since all experiments run in a cloud environment, we additionally report cost, decomposed into two components: (i) compute cost, determined by the hourly price of the instance, and (ii) storage-access cost, determined by the number of S3 requests issued. The rate for the **m6idn.16xlarge** instance is \$5.092 per hour, and the cost for S3 is \$0.0004 per 1,000 requests.

The total cost of a query is computed as:

$$\text{Cost} = \underbrace{r_{\text{S3}} \cdot \frac{N_{\text{req}}}{1000}}_{\text{S3 cost}} + \underbrace{r_{\text{compute}} \cdot \frac{t_{\text{exec}}}{3600}}_{\text{compute cost}}, \quad (5.1)$$

where r_{S3} is the S3 price per 1,000 requests, N_{req} is the number of S3 requests issued by the query, r_{compute} is the hourly instance rate, and t_{exec} is the query execution time in seconds. For example, a query that issues 15,000 S3 requests and runs for 10 seconds costs $0.0004 \cdot 15 + 5.092 \cdot \frac{10}{3600} \approx \$0.020$, of which \$0.006 is attributable to S3 requests and \$0.014 to compute.

5.3 TPC-H

In this section, we evaluate TPC-H at SF=500, building up the read path one optimization at a time, from a simple synchronous read through caching to prefetching, and measuring the cost of each. Unless stated otherwise, every number is a cold run on a **m6idn.16xlarge** instance.

5. PERFORMANCE EVALUATION

	GM	Total	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Cold (local disk)	9.30	275.25	25.09	1.20	13.97	7.55	14.34	10.53	18.52	19.22	28.24	12.56	1.15	9.82	9.73	15.60	13.38	1.47	12.82	11.17	14.14	16.59	17.49	1.67
Warm (in-memory)	1.90	61.35	2.25	0.43	2.38	2.00	2.31	0.59	2.00	1.75	8.65	3.39	0.33	1.86	8.79	1.39	1.11	1.30	1.59	7.50	1.29	1.48	7.86	1.10
Factor ($\times$)	4.89	4.50	11.15	2.79	5.87	3.77	6.21	17.85	9.26	10.98	3.26	3.71	3.48	5.28	1.11	11.22	12.05	1.13	8.06	1.49	10.96	11.21	2.23	1.52

Table 5.2: TPC-H SF=500 runtime (seconds) using vanilla DuckDB on the AWS NVMe SSD instance (`m6idn.16xlarge`): cold (local disk) vs. warm (in-memory). GM is the geometric mean across all queries; Factor is the speedup.

5.3.1 Vanilla DuckDB

We first develop a sense of how DuckDB performs under naive schemes, which sets the two reference points we will chase: cloud-based local disk and vanilla S3.

Vanilla DuckDB on NVMe SSD. Table 5.2 shows the result on the EC2 instance with local storage. We benchmark I/O performance with `fio` and see a maximum throughput of 2.5 GiB/s. The gap between executing with data in memory and from local disk is roughly fivefold. This makes clear that I/O is the bottleneck: if we can raise the effective read throughput, performance will jump.

Vanilla DuckDB on S3. DuckDB can attach the single database file directly from S3 with a simple `ATTACH 's3://my_bucket/tpch-sf500' AS db`. The result is shown in Figure 5.1. Reading the same DuckDB file over `httpfs`, with neither a caching layer nor prefetching, is about $6\times$ slower than local disk (1666 s against 275 s), since it pays the full S3 round-trip on the read path of every scan.

5.3.2 Synchronous read without caching, on the single-buffer and per-column layouts

In this section, we move on to our own extension implemented with differential storage. In this setting, we do not add any optimization; we use our extension as the intermediate layer to serve DuckDB file system requests to S3. Every read is issued synchronously using the mapping. DuckDB first requests a logical byte range. It then calls the file system in the extension to look up in the mapping and locates the corresponding S3 objects. Finally, it fetches them and returns the bytes. Caching and prefetching are not enabled, so the results are a lower bound before any of the optimizations from Chapter 4 are turned on.

Overview. Figure 5.1 shows the five configurations side by side, reporting the total cold execution time over all 22 TPC-H queries. Vanilla DuckDB on local NVMe is the fastest at

275 s, since requests are served locally. Vanilla DuckDB that reads its single-file database from S3 over the `httpfs` extension is the slowest at 1666 s, roughly $6\times$ worse than local. It has neither a caching layer nor any prefetching logic and pays the full S3 round-trip on every read. With the extension, this naive synchronous mode lands below vanilla S3: 1147 s on the per-column layout with the default row group size (`rgsx1`) and 995 s with the larger row group size (`rgsx4`). Increasing the row group size improves query performance.

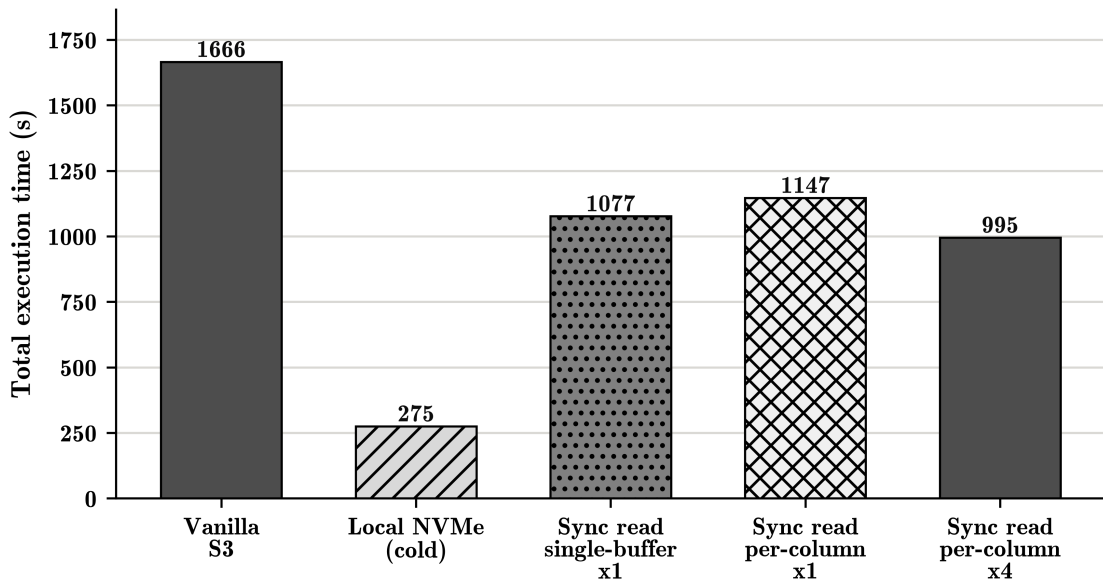


Figure 5.1: Total cold execution time over the 22 TPC-H SF=500 queries, one bar per use case. Vanilla S3 is the slowest; local NVMe is the target.

Single-buffer versus per-column The two write layouts land close to each other in this bare synchronous mode. Still, single-buffer is the faster of the two, 1077 s against per-column’s 1147 s, because a naive remote read passes through *two* layers of coalescing and only the write layout decides whether the second one fires. The first layer is DuckDB’s own. When DuckDB stores a table, each column within a row group occupies one or more 256 KiB blocks, and those blocks are likely assigned *logically consecutive* block IDs at write time. For a remote database, DuckDB runs a *prefetch* step: before scanning a row group, it gathers the block IDs it will touch. It merges consecutive blocks into a few large read *ranges* instead of sending one. This is *logical* coalescing, decided by block IDs alone.

The second layer is in the direct read path (`ReadMode::DIRECT`) of our extension. Each range of bytes, containing multiple logical blocks, that the logical prefetch asks for is mapped to some (object, offset) pair on S3. Our read path then merges the blocks that land

5. PERFORMANCE EVALUATION

physically consecutive within one object into a single large GET. This is *opportunistic*: it only merges blocks with consecutive IDs a scan has already asked for in the DuckDB logical view that happen to be also adjacent on the physical layout, and it is a different mechanism from the coalescing performed by the batch prefetcher we introduce later, which collects and reorders a whole batch of upcoming row groups' blocks to build large requests on the physical layout. In a single-file database on local disk, a block ID is an offset into the file ($\text{block_id} \times 256 \text{ KiB}$), so logical contiguity always implies physical contiguity. Differential storage breaks that, so whether a logically consecutive run is still one contiguous byte range now depends on where the writer placed the blocks. The single-buffer writer appends blocks in flush order, so a row group's consecutive IDs likely remain physically contiguous within a single 16 MiB object; however, the two-layer coalescing only merges a small share of fetches (1.06 blocks per request). The per-column writer routes each column's blocks to its own buffer, so the logical layer still merges the request but the physical layer cannot, and almost nothing coalesces (1.00 block per request).

This physical coalescing is mechanical: it merges block IDs, not what the query reads. The block ID adjacency is not what a query needs. Per-column layout instead creates *semantic* locality: it clusters the data a query actually reads. As a query touches only a few of a table's columns, keeping those columns' blocks together in one object is what caching and prefetching can benefit from.

The effect of row group size. Increasing the row group size (rgsx4) brings the workload down from 1147s to 995s, a $1.15\times$ speedup (Figure 5.1). The total stored footprint is essentially unchanged, 129.8 GiB at rgsx1 against 127.7 GiB at rgsx4, so a larger row group compresses only marginally better. What changes is the layout. As shown in Section 4.2.2, a larger row group makes each column-row-group fill whole blocks, cutting the share of multi-segment blocks from 36.7% to 6.4%. Fewer multi-segment blocks means a per-column scan pulls in far less co-located data from other columns, so it reads fewer bytes for the same query: rgsx4 reads only 455 GiB over the workload against rgsx1's 599 GiB, $1.32\times$ less (Figure 5.2).

5.3.3 Synchronous read with caching, on the single-buffer layout

We now add a cache. On any read, in addition to serving the requested 256 KiB block, we fetch the entire 16 MiB object into a local disk cache (NVMe SSD), so that subsequent reads of the same object are served from the cache. The fetch runs in one of two modes: *synchronous* caching, where the read waits for the whole 16 MiB to arrive, and *asynchronous*

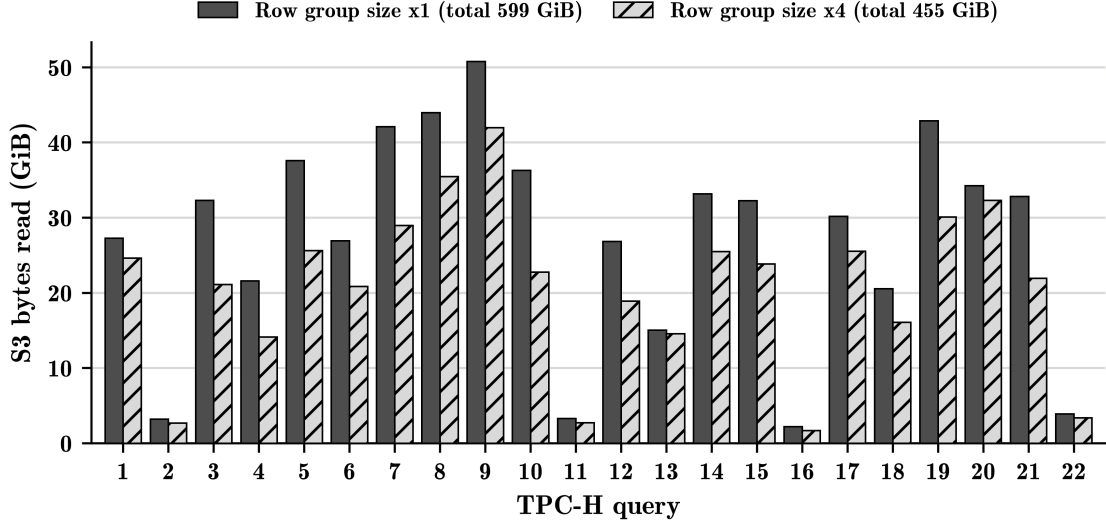


Figure 5.2: Per-query S3 read volume (GiB) for rgx1 (599 GiB total) versus rgx4 (455 GiB total). Changing the row group size reads $1.32\times$ fewer bytes over the workload.

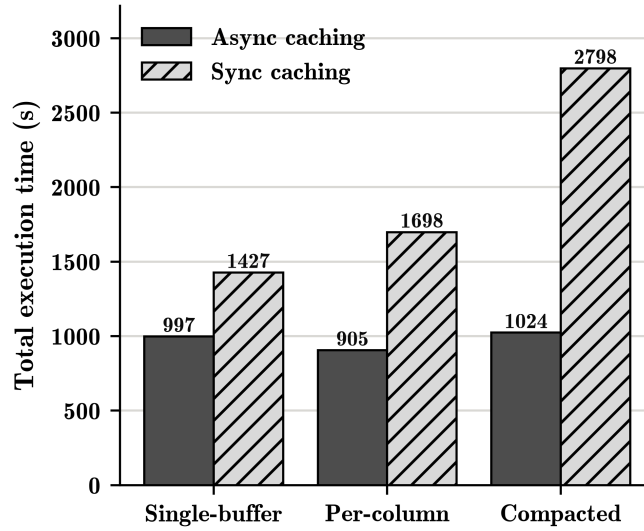


Figure 5.3: Synchronous versus asynchronous caching (no reorderer), total cold time by write layout. Synchronous caching makes every read wait for a full 16 MiB transfer, so it is consistently slower.

caching, where the 256 KiB block is served immediately, and the 16 MiB fetch runs in the background. All results below use the asynchronous mode unless stated otherwise, because it keeps the 16 MiB fetch off the critical path; Figure 5.3 shows synchronous caching is consistently slower, and much slower once the layout is clean (2798s versus

5. PERFORMANCE EVALUATION

Write layout	Synchronous	Asynchronous
Single-buffer	1802	1931
Per-column	1312	1662
Compacted	595	1031

Table 5.3: Total S3 bytes read (GiB) under caching, by write layout and mode (TPC-H SF=500, rgxs1, no reorderer). Synchronous caching fetches only the 16 MiB objects; asynchronous caching adds a 256 KiB read per miss. A single cold pass needs 599 GiB.

1024s on the compacted layout), since every read blocks on a full 16 MiB transfer. It is counter-intuitive that performance is worst on the *cleanest* layout. The cause is DuckDB’s sequential row-group hand-out. In synchronous mode, threads wait on the same object at the same time, blocking until that whole 16 MiB object has arrived in the disk cache. The entire thread pool stalls together on a single large transfer instead of overlapping many. A mixed layout, i.e., a single-buffer layout, spreads the threads across different objects, so their stalls overlap and the penalty is smaller.

The two modes also read different volumes (Table 5.3). Synchronous caching issues only the 16 MiB object fetches and serves each block from the object it waited for, so it reads the least, on compacted just 595 GiB, essentially the volume a single cold pass needs (599 GiB). Asynchronous caching additionally requests immediately a 256 KiB read for each miss while the 16 MiB fetch runs in the background. As a result, it fetches the same objects plus a few hundred GiB of redundant small reads. Either way, the 16 MiB fetch volume, and hence the wasted bytes, shrinks with a cleaner layout; asynchronous caching pays some extra bandwidth and more requests to buy the latency that makes it the faster mode.

Figure 5.4 is the big picture for the rest of this chapter: the total cold time for the three write layouts, with and without the reorderer, against the no-cache baseline (1147s) and local NVMe (275s). We walk through it layout by layout.

On the single-buffer layout, caching barely helps: the workload runs in 997s, only marginally below the 1147s no-cache baseline. The reason is the layout. As a single-buffer object mixes different columns into S3 files, a 16 MiB fetch triggered by a single 256 KiB block contains data the query will never read. The cache hit rate stays at just 21% (Figure 5.5) while the workload pulls a wasteful 1931 GiB off S3 (Figure 5.6). Caching a mixed object is close to caching noise: even with the reorderer, a single-buffer object only reaches a 45% hit rate, against 90% on the compacted layout. TPC-DS shows the same

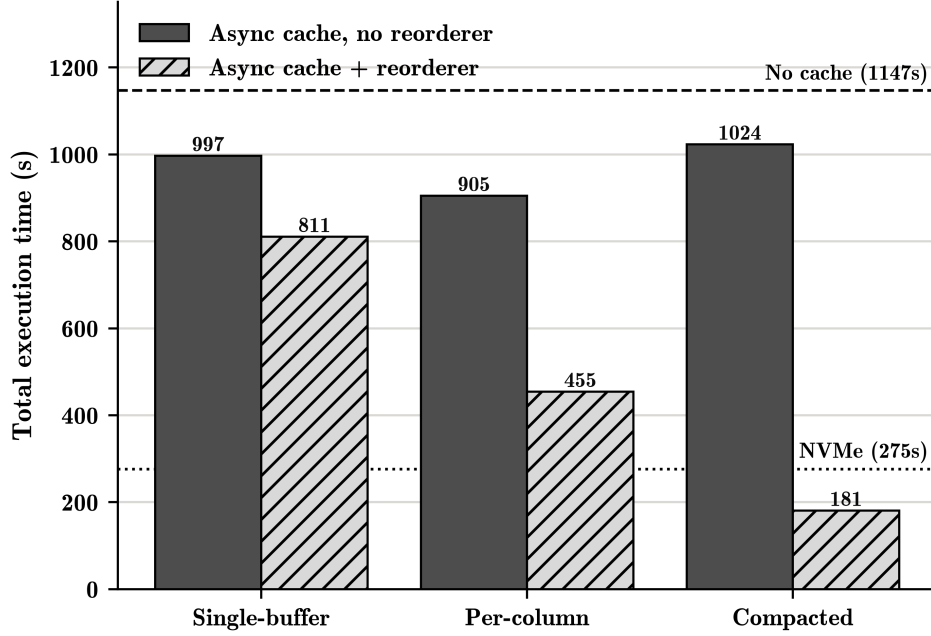


Figure 5.4: Total cold execution time over the 22 TPC-H SF=500 queries, asynchronous caching, for the three write layouts, with and without the modulo reordering. Dashed line: S3 no-cache baseline (1147s); dotted line: vanilla DuckDB on local NVMe (275s). Caching alone barely moves the baseline; the reordering is what unlocks it, and the compacted layout with the reordering beats local NVMe.

pattern more sharply, as Section 5.4 discusses.

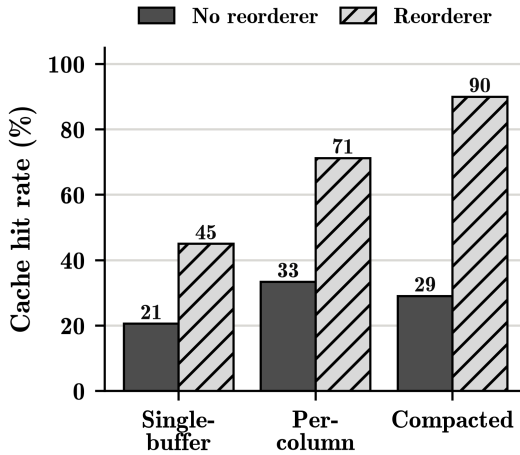


Figure 5.5: Cache hit rate by write layout, asynchronous caching, with and without the reordering.

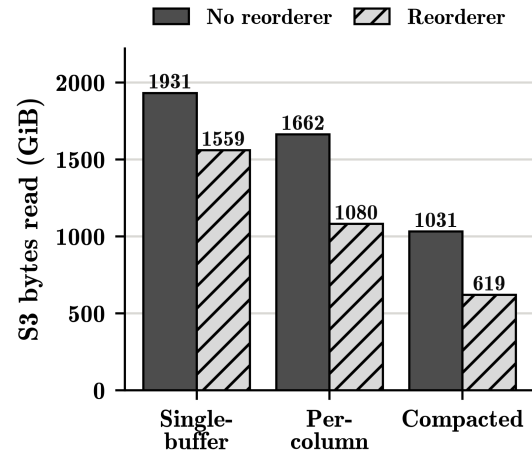


Figure 5.6: S3 bytes read over the workload by write layout, asynchronous caching, with and without the reordering.

5. PERFORMANCE EVALUATION

5.3.4 Synchronous read with caching, on the per-column and compacted layouts

We expect that putting a column’s blocks together should make caching pay off. The reason is that a 16 MiB object now contains useful bytes from a scanned column. The bytes read confirm the layout is cleaner. The per-column writer reduces the workload’s S3 read volume from 1931 GiB on single-buffer layout to 1662 GiB. The compacted layout, which groups blocks by their full column combination, cuts it further to 1031 GiB (Figure 5.6). A block read now brings in an object that is actually needed.

However, the wall-clock does not improve: 905s for per-column and 1024s for compacted. Both are still near the baseline. The reason is the order in which row groups are dispatched. DuckDB hands out row groups almost sequentially, and consecutive row groups sit close together on S3. As a result, all threads target the same object at once and then move on all together. The object is consumed immediately rather than reused later, so the hit rate stays low (33% per-column, 29% compacted; Figure 5.5) and the cleaner layout brings no improvement. The compacted layout even runs slightly slower here despite reading the fewest bytes.

5.3.5 Synchronous read with caching and the modulo reorderer

The last optimization is the modulo reorderer, which breaks the sequential dispatching by splitting row groups by their ID modulo a constant and serving one group at a time. Hence, at any given moment, the threads work on row groups that reside in different objects, and an object fetched now is reused a few steps later. This is where caching finally pays off, because the cached objects are now both useful (per-column) and reusable (via strided access). The hit rate rises, but only where the layout can support it (Figure 5.5): to 90% on compacted and 71% on per-column, against just 45% on single-buffer. The total time follows (Figure 5.4): single-buffer to 811s, per-column to 455s, and compacted to 181s. Only now does the layout performance ordering come through cleanly: (1) compacted; (2) per column; (3) single-buffer.

We can see that the compacted layout with the reorderer delivers faster total execution time than vanilla DuckDB on local NVMe (181s versus 275s). Two effects can explain it. First, the aggregate S3 network bandwidth (11.6 GiB/s) exceeds that of a single local NVMe SSD (2.5 GiB/s). A well-pipelined remote read can perform better than a local disk. The second effect is the OS page cache. A single local SSD is not only the write target for objects from S3, but also the read source. On one device, reads and writes compete for the

same bandwidth. Because we do not use `O_DIRECT`, both paths go through RAM. S3 files are buffered in memory and flushed to disk lazily. Recently cached objects then stay in memory and can be read directly. Moreover, the OS page cache is a write-back buffer that hides the disk's write and keeps the read and write paths from contending. With `O_DIRECT` enabled, writes and reads both compete on the one disk. For these reasons, caching turns S3 into an advantage once the layout is clean enough to keep the hit rate high.

5.3.6 Prefetching

Prefetching attacks the latency more directly and on the read path itself. Instead of waiting for a block when a thread asks for it, we look ahead in the scan, determine which blocks the upcoming row groups will need, and issue those S3 requests early into the in-memory buffers, so that by the time a thread reaches a row group, its blocks have already arrived. Moreover, with prefetching, we have complete control of how many concurrent requests we want in the system. All results in this section run on the direct (no-cache) read path with the per-column and compacted layouts, at both the default row group size (`rgsx1`) and the $4\times$ size (`rgsx4`), TPC-H SF=500, cold.

5.3.6.1 Normal prefetching

The first scheme prefetches the blocks of the upcoming row groups one at a time, issuing one 256 KiB S3 request per block ahead of the scan. We maintain 1024 concurrent requests to balance the thread pool sizes for handling these requests and the CPU time spent on query processing. Even this naive look-ahead is a large win over everything so far: the workload finishes in 167 s on the per-column layout (Figure 5.7), already below both local NVMe (275 s) and the best caching configuration (181 s), because a thread rarely blocks on a read. Increasing the row group to `rgsx4` helps further, to 147 s, since it reads fewer bytes (455 GiB against 599 GiB, the multi-segment reduction of Section 4.2.2). But the request pattern is poor regardless of layout or row group size: prefetching every block individually issues 2.44 million (`rgsx1`) or 1.86 million (`rgsx4`) S3 requests over the workload, each only 256 KiB, so the aggregate S3 throughput is capped at about 3.6 GiB/s on average (Figure 5.10), under what the network can deliver, matching the small-request ceiling identified in the object-storage analysis.

5. PERFORMANCE EVALUATION

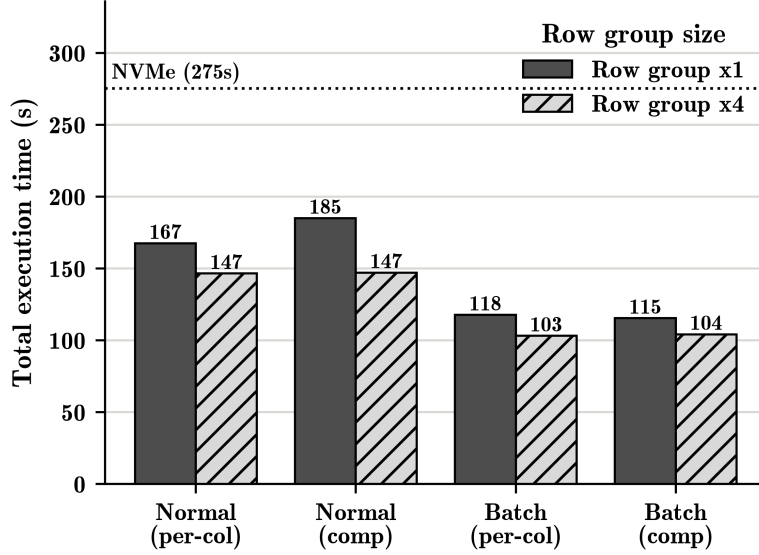


Figure 5.7: Total cold execution time for normal (row-group) prefetching and batch prefetching, on the per-column and compacted layouts, at row group size x1 and x4. Dotted line: vanilla DuckDB on local NVMe read cold (275 s). Both prefetch schemes run on the direct, no-cache path.

5.3.6.2 Batch prefetching

Batch prefetching keeps the same look-ahead but coalesces it. It takes a batch of upcoming and consecutive row groups, collects the blocks they will read, sorts those blocks by their physical positions on S3, and merges contiguous blocks within the same object into a single large GET. We fix the *batch size*, the number of row groups gathered per coalescing round, at 256: a larger batch coalesces across more row groups into fewer, bigger requests but spends longer collecting a round before dispatching it and keeps fewer requests in flight (we cap the number of concurrent requests to 512). The coalescing effect depends sharply on both the layout and the row group size (Figure 5.8). On the per-column layout, where multi-segment blocks fragment a column’s runs, coalescing merges 3.7 blocks per request at rgsx1 and 12.9 at rgsx4 (the larger row group removes most of those multi-segment blocks). The compacted layout is already clean, each object holding an ordered run of one column combination, so it merges 24 blocks per request at rgsx1 and 36 at rgsx4. The request count over the workload drops from the 2.44 million individual GETs of normal prefetching to 102,000 (rgsx1) and 51,000 (rgsx4) on compacted, a 24–36 $\times$ reduction.

Blocks per request understates how well the bytes are packed, because a scan still issues a few small requests that drag the simple average down. Weighting each request by the bytes

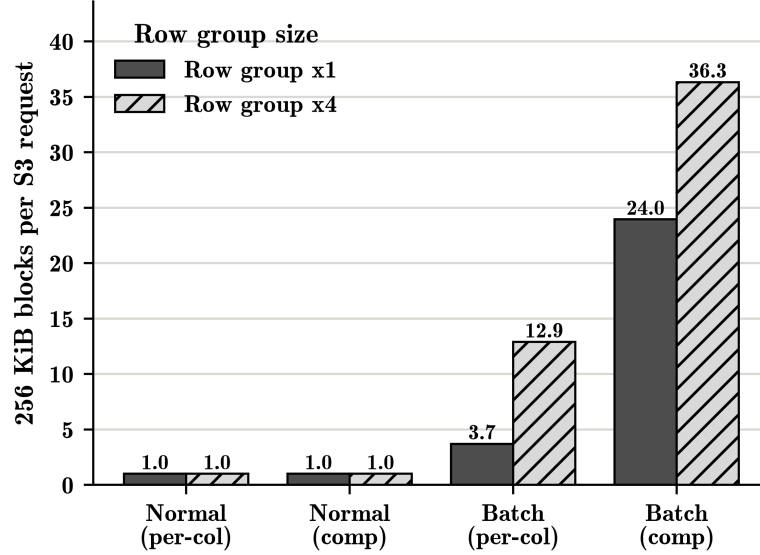


Figure 5.8: Coalescing efficiency of batch prefetching (batch size 256): 256 KiB blocks merged into a single S3 request, by layout and row group size. Normal prefetching issues one block per request; batch prefetching on the compacted layout reaches 24–36 blocks per request, while on the per-column layout a larger row group increases it from 3.7 (rgsx1) to 12.9 (rgsx4) as multi-segment blocks disappear.

it moves tells the real story (Figure 5.9): the byte-weighted request size lands in or just below the 8–16 MiB range S3 serves best, 12.7–15.1 MiB on compacted and 6.8–12.7 MiB on per column, even though the simple average on per-column is only 0.9–3.2 MiB. The bulk of the data always arrives via large GET requests.

Larger requests increase throughput: batch prefetching sustains up to 5.1 GiB/s on average and peaks at 9.2 GiB/s across queries, against 3.6 average and 5.8 peak for normal prefetching (Figure 5.10). The wall-clock follows: the batch configurations are the fastest we measure, 103–118 s, with rgxs4 showing better execution time. Coalescing maximizes request size and minimizes cost. However, batch prefetching adds only a modest further speedup, about 1.4 $\times$ (147 s $\rightarrow$ 104 s on compacted rgxs4, a 30% gain). Its larger gain is on cost, which the next subsection quantifies.

5.3.6.3 Cost

The cost of a workload run is divided into the S3-request charge (\$0.0004 per 1,000 requests) and the compute charge (the `m6idn.16xlarge` rate of \$5.092 per hour applied to the execution time), following the cost model of Equation (5.1). Figure 5.11 shows the breakdown of costs across configurations.

5. PERFORMANCE EVALUATION

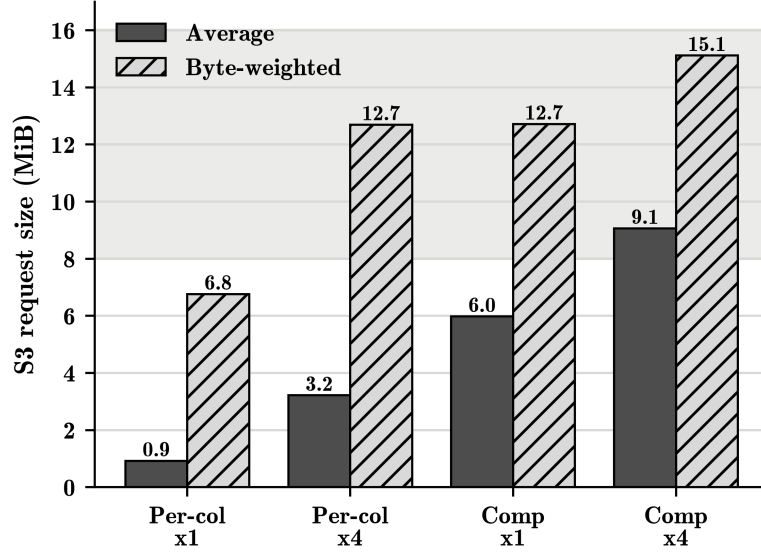


Figure 5.9: Batch prefetching (batch size 256) S3 request size, simple average versus byte-weighted, by layout and row group size. The shaded band is the 8–16 MiB range S3 serves best. Even where the simple average is small (per-column), the byte-weighted size shows most bytes arrive in large requests.

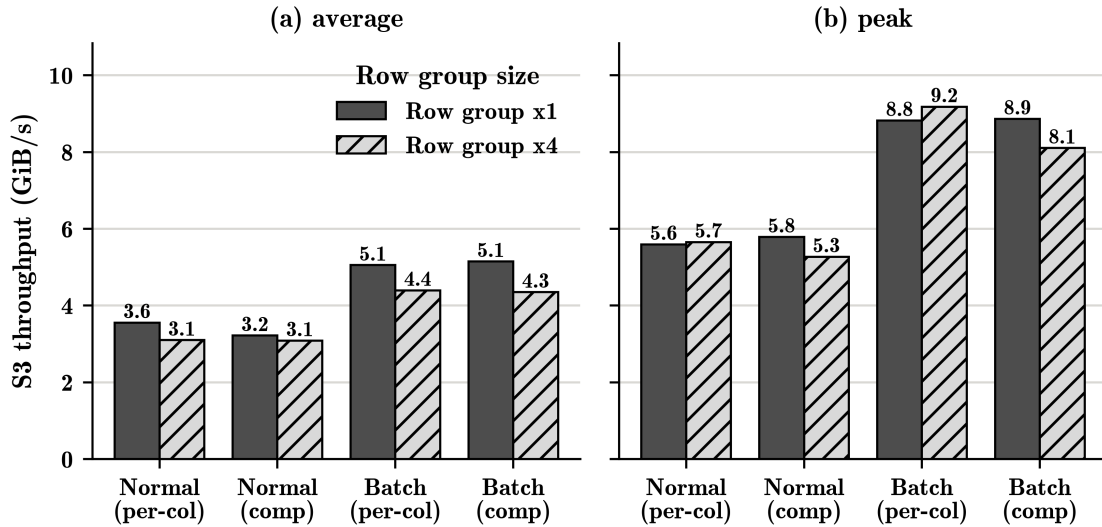


Figure 5.10: Average (a) and peak (b) S3 throughput across the 22 queries, for normal and batch prefetching on both layouts and row group sizes. Coalescing into larger requests raises both the average and the peak throughput.

In normal prefetching, the S3 request cost is the primary factor: millions of 256 KiB GETs result in an S3 charge of \$0.98 (rgsx1), compared to only about \$0.24 for compute.

Setting the row group size to rgx4 reduces the S3 charge to \$0.95, but the costs remain dominated by the request count. The coalescing effect of batch prefetching significantly reduces the S3 charge to \$0.04 (rgsx1) and \$0.02 (rgsx4). The total cost then decreases to \$0.20 and \$0.17, a sixfold reduction. Coalescing therefore provides two benefits: it increases throughput and shifts the workload away from per-request pricing, which is the primary contributor to S3 access costs.

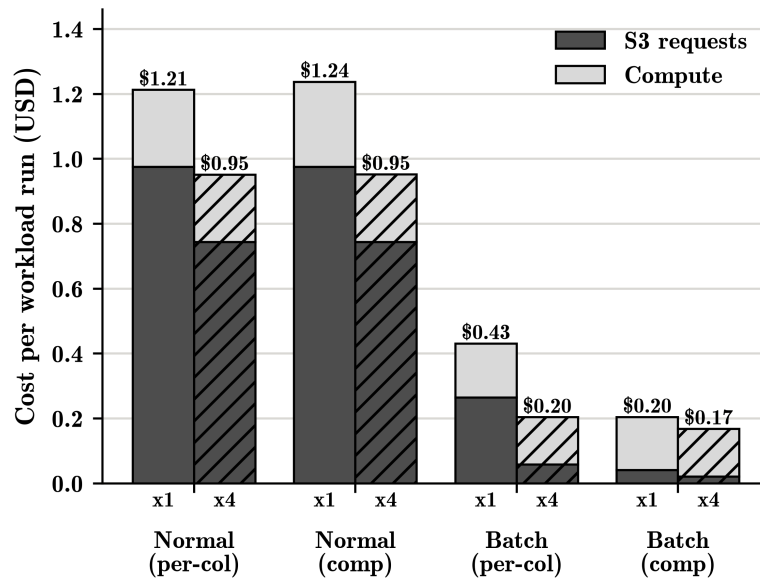


Figure 5.11: Cost of one TPC-H SF=500 workload run, split into S3-request and compute costs, by layout and row group size (x1, x4). Normal prefetching is dominated by the request charge for its millions of 256 KiB GETs; batch prefetching coalesces these, leaving compute as the main cost and cutting the total by up to 7×.

5.3.7 Putting it all together

We can now step back and validate, against the measurements, the hypotheses laid out in the design chapter (Chapter 4). Figure 5.12 draws the whole picture: the total cold TPC-H SF=500 time for the best configuration at each stage, from vanilla DuckDB on S3, through synchronous reads with caching, to prefetching, at both row group sizes, with local NVMe as the reference we set out to beat.

The whole picture. Vanilla DuckDB reading its database file from S3 takes 1666s. Simply routing reads through differential storage, without optimization, reduces this to 1147s. Adding a cache is worthless on its own, but a clean (compacted) layout together

5. PERFORMANCE EVALUATION

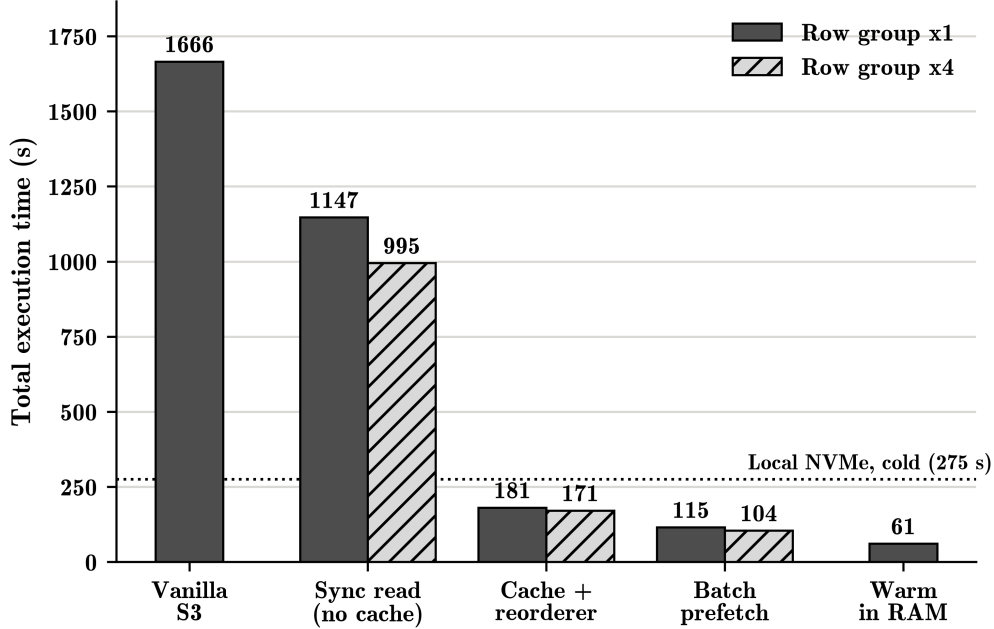


Figure 5.12: The whole picture. Total cold execution time for the best configuration at each optimization stage, row group size x1 versus x4. Dotted line: vanilla DuckDB on local NVMe read cold (275 s). Each stage adds one idea from the design: routing through differential storage, then caching with a clean layout and the reorderer, then batch prefetching. The last bar is the floor: the same engine with the data already warm in the buffer pool (61 s), doing no I/O at all. The best configuration is 16× faster than vanilla S3 and below local disk.

with the modulo reorderer brings the workload to 181 s, already below local NVMe (275 s). Prefetching removes the last of the read latency from the critical path: batch prefetching on the compacted layout at rgsx4 reaches 104 s, with per-column at rgsx4 effectively tied at 103 s. End-to-end, the best configuration is about 16× faster than vanilla S3 and 2.6× faster than a local NVMe SSD, confirming the hypothesis of this thesis: with the right layout and read path, executing DuckDB against cloud object storage can be efficient. That is only 1.7× slower than the same engine running entirely in memory, with no I/O at all (104 against 61 s; Table 5.2).

Validating the hypotheses. Each design decision behaves as predicted:

- **Layout mixing hurts, per-column helps.** The single-buffer layout mixes a median of 16 columns per object; caching it wastes bandwidth, and its hit rate stays at 21%. Per-column and compacted layouts read far fewer bytes (from 1931 to 1031 GiB under caching), as expected.

- **Caching needs the reorderer.** Caching alone barely moves the baseline (from 1147 to 905–1024s) because sequential row-group hand-out consumes each cached object immediately. The modulo reorderer restores reuse, increasing the hit rate to 45–90% and collapsing the time to 181s on the compacted layout.
- **Larger row groups clean up the layout.** Increasing the row group to rgsx4 cuts multi-segment blocks from 36.7% to 6.4%, which results in fewer wasted bytes in synchronous reads ($1.32\times$), and better coalescing in prefetching (per-column jumps from 3.7 to 12.9 blocks per request).
- **Prefetching beats caching, and coalescing hits the S3 sweet spot.** Even naive prefetching (167s) beats the best caching configuration, and batch prefetching moves most of its bytes in 13–15 MiB (byte-weighted) requests, falling in the 8–16 MiB range S3 serves best, sustaining up to 9.2 GiB/s peak throughput.

The role of caching. A deployment would normally place several layers of cache between the engine and storage. Immutability makes them easy to add, as a copy never has to be invalidated. We use a local NVMe disk in our experiments to keep recently fetched objects from S3. We can go further and dedicate a pool of SSDs as a longer-lived cache, shared across sessions. Our results suggest that the long-lived caching tier is harder to justify than it first appears. A TPC-H run costs \$0.17 with prefetching alone, which is both the fastest and the cheapest. A dedicated cache pool, by contrast, is paid for whether or not it is being used. The cheaper approach is only having a local SSD cache, with cold-read latency handled by either prefetching or caching.

Wrapping up cost. Cost is the second axis of the whole picture. It does not simply track wall clock time. We price each run with the model of Equation (5.1). Figure 5.13 traces the cost across different configurations.

The synchronous no-cache configuration is the most expensive at \$2.64, being both slow (\$1.66 of compute) and issuing 2.45 million GETs (a further \$0.98 in requests). Caching and batch prefetching coalesce those requests, with caching at \$0.31 and batch prefetching on compacted rgsx4 at \$0.17. From worst to best, the cost drops about $15\times$ with an $11\times$ wall-clock gain, because coalescing removes the per-request cost that sat on top of compute.

5. PERFORMANCE EVALUATION

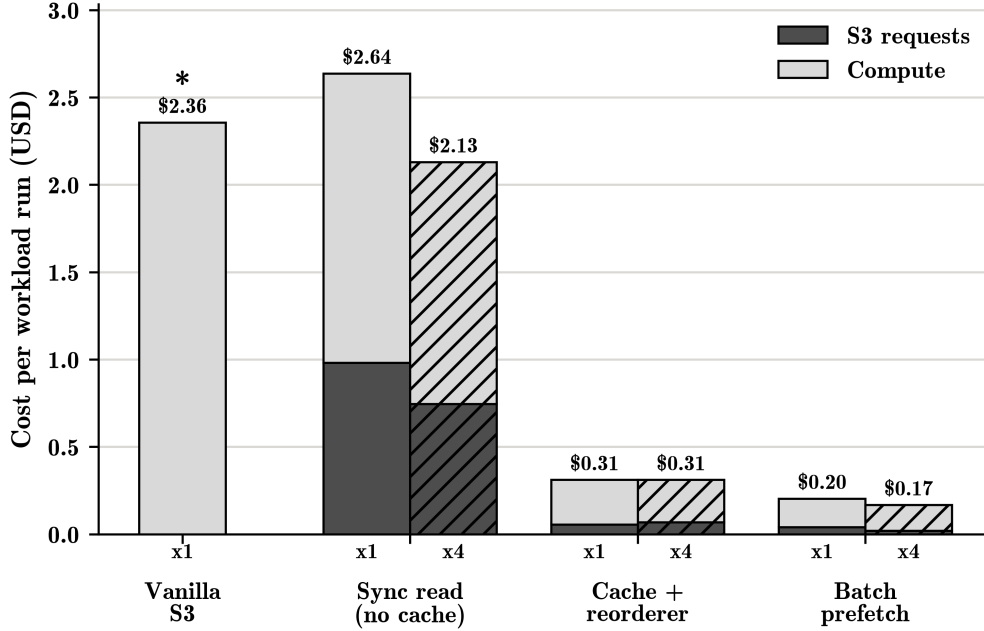


Figure 5.13: Cost of one TPC-H SF=500 workload run across the optimization journey, split into S3-request and compute charges, at row group size x1 and x4. Fine-grained access (synchronous, no cache) is the most expensive, paying both high compute and a large request charge; coalescing under caching and then prefetching removes the request charge and, with it, most of the cost. * Vanilla S3 runs on plain DuckDB, whose S3-request count is not instrumented, so only its compute component is shown.

5.4 TPC-DS: does the trend hold?

The TPC-H analysis established a clear positioning: a clean per-column layout plus the modulo reorderer makes caching pay off, prefetching with coalescing does better still, and batch prefetching on a compacted, large-row-group layout is both the fastest and the cheapest configuration. TPC-DS is a more demanding workload, 99 complex queries over a 24-table schema, and it is more compute-heavy than TPC-H, so it is a good stress test of whether these conclusions are specific to TPC-H or general.

5.4.1 Differential storage versus vanilla S3

We start where TPC-H did: does simply routing reads through the differential-storage mapping, before any caching or prefetching, perform better than vanilla DuckDB reading its database file from S3? Vanilla DuckDB over https runs the 99 queries in 2533s (geometric mean 15.3s). Our extension in bare synchronous mode (direct reads, no cache, no prefetch

optimization) on the default single-buffer layout runs in 1969 s (geometric mean 11.9 s), 1.29× faster than vanilla S3 on both the total and the geometric mean.

Single-buffer versus per-column TPC-DS exercises the two-layer logical-to-physical coalescing from the TPC-H analysis even better. Single-buffer layout, whose flush-order blocks keep a row group’s IDs physically together, coalesces to 1.35 blocks per request (against TPC-H’s 1.06), so its 1969 s is better than both vanilla S3 and per-column. Per-column scatters each column into its own object, so nothing merges (1.00 block per request, 0%) and it is slower than vanilla, at 3023 s.

5.4.2 Synchronous read with caching

We add the same optimistic cache and benchmark on the three write layouts, with and without the modulo reorderer. Figure 5.14 reports the total time, the geometric-mean per-query time, the S3 bytes read, and the cache-hit rate.

The results tell the same story we had seen from TPC-H. Caching without the reorderer does not even perform better than the naive read. No-reorderer configurations land between 2154 and 2374 s. The hit rate stays at 12–20%. Turning on the reorderer increases the hit rate to 95% on compacted and 93% on per-column. We also reduce the time with the reorders: compacted drops to 449 s, per-column to 507 s, and single-buffer only to 1458 s. The ordering compacted–per-column–single-buffer is exactly as on TPC-H, and it tracks the S3 bytes each reads ($1325 < 1715 \ll 4208$ GiB): the cleaner the layout, the more useful each cached object and the fewer bytes wasted. The geometric mean shows the same story.

5.4.3 Prefetching

With prefetching enabled on the read path, Figure 5.15 compares normal prefetching against batch prefetching (default batch size 256) on the per-column and compacted layouts at both row group sizes.

Batch prefetching coalesces exactly as it did on TPC-H. On the compacted layout, it merges 36–40 blocks into each S3 request, cutting the request count from normal prefetching’s 4.6 million (rgsx1) and 3.1 million (rgsx4) individual 256 KiB GETs down to 127,000 and 78,000 respectively; on the per-column layout, where multi-segment blocks fragment the runs, it merges only 6–7 blocks. Batch prefetching is the fastest configuration we measure on TPC-DS, at 375 s with compacted at rgxs4, compared to 462–599 s for normal prefetching.

5. PERFORMANCE EVALUATION

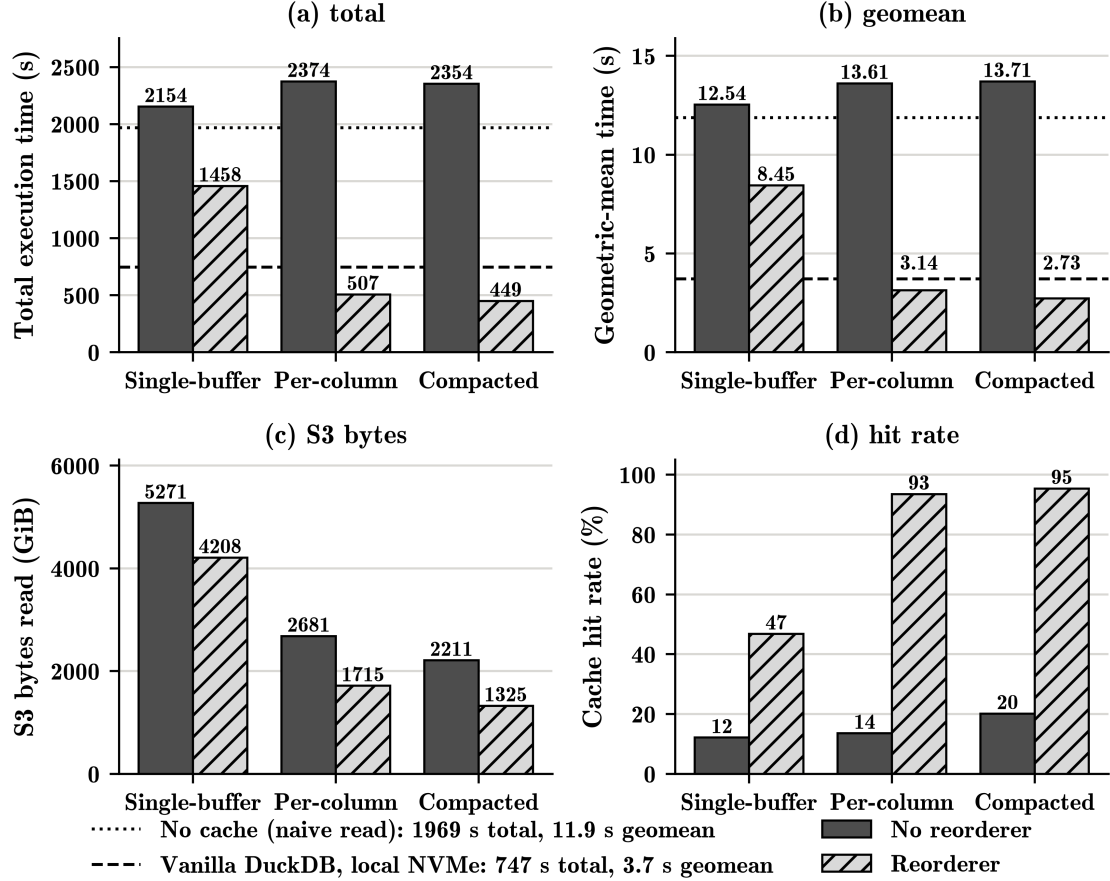


Figure 5.14: TPC-DS SF=500, asynchronous caching, by write layout with and without the reorderer: (a) total time, (b) geometric-mean time, (c) S3 bytes read, (d) cache-hit rate. The dotted line marks the no-cache baseline, and the dashed line is vanilla DuckDB on local NVMe. As in the TPC-H benchmark, simple caching does not really improve from the baseline; the reorderer unlocks it.

The byte-weighted request size (Figure 5.17) again lands in the 8–16 MiB range S3 serves best: 14.1–14.9 MiB on compacted and 9.8–11.2 MiB even on per-column, whose simple average is only 1.5–1.7 MiB. As on TPC-H, the bulk of the data arrives in large GETs.

The biggest TPC-H/TPC-DS difference: on TPC-DS, *normal* prefetching (462–599 s) is no better than caching with the reorderer (449–507 s), whereas on TPC-H even normal prefetching beat caching. Only *batch* prefetching is the clear winner here. In addition, the execution time gain from coalescing is accordingly modest: batch prefetching is only about $1.2\times$ faster than normal prefetching at the same config (462 s $\rightarrow$ 375 s on compacted rgx4, a $\sim 20\%$ gain); its larger gain is again cost, cutting the request count 36–40 $\times$ and the cost from \$1.90 to \$0.56.

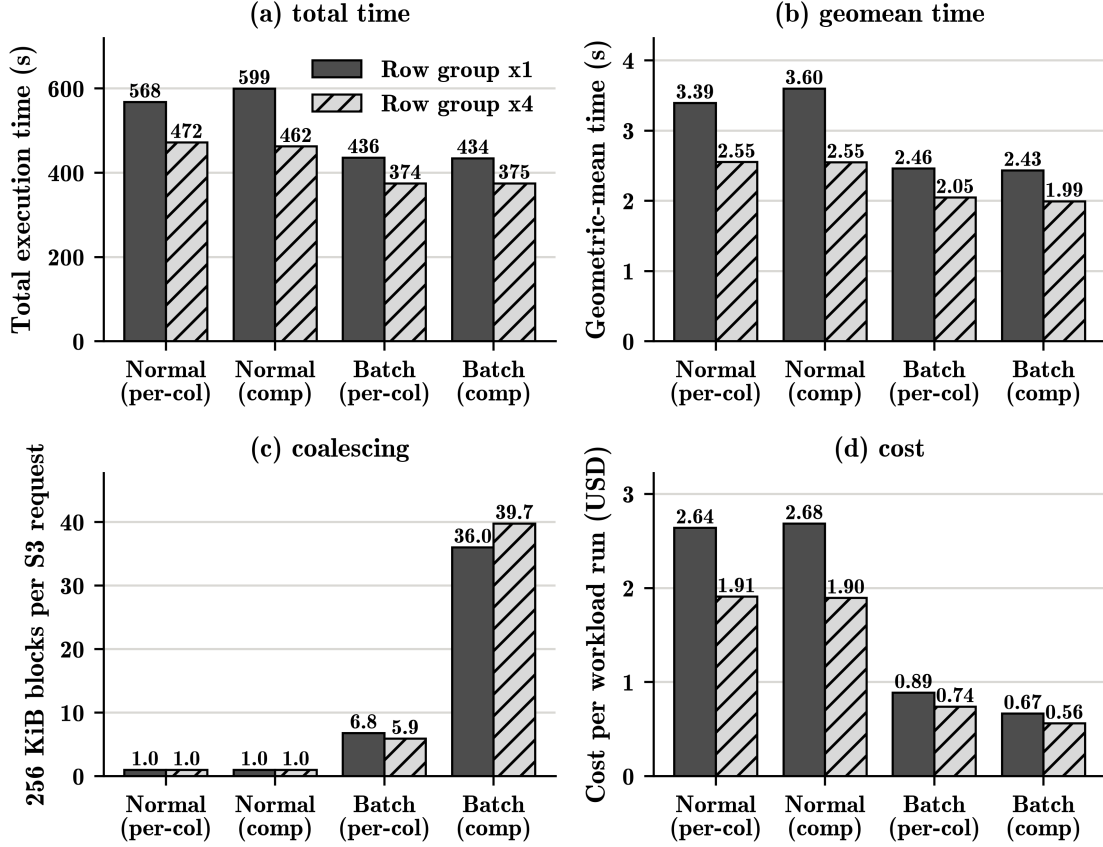


Figure 5.15: TPC-DS prefetching on the direct (no-cache) path: (a) total time, (b) geometric-mean time, (c) coalescing (256 KiB blocks per S3 request), (d) cost, by layout and row group size. Batch prefetching coalesces the stream of small requests into large ones and is the fastest and cheapest.

5.4.4 Putting it all together

Figure 5.18 draws the whole journey, total time and geometric mean, and Figure 5.19 the cost, both against local NVMe.

The progression matches TPC-H stage for stage. The naive synchronous no-cache read on the default single-buffer layout takes 1969s; caching with the reorderer on a compacted layout brings this to 449s, already below local NVMe (747s); and batch prefetching reaches 375s, a $2.0\times$ speedup over local disk and a $5.3\times$ speedup over the naive synchronous read. That is only $1.8\times$ slower than the same engine running entirely in memory (375 against 213s). Increasing the row group (rgsx4) helps at the synchronous-read and prefetching stages, but unlike on TPC-H it does not help caching with the reorderer, which is slightly slower and more expensive at rgx4 (Figure 5.18, Figure 5.19).

5. PERFORMANCE EVALUATION

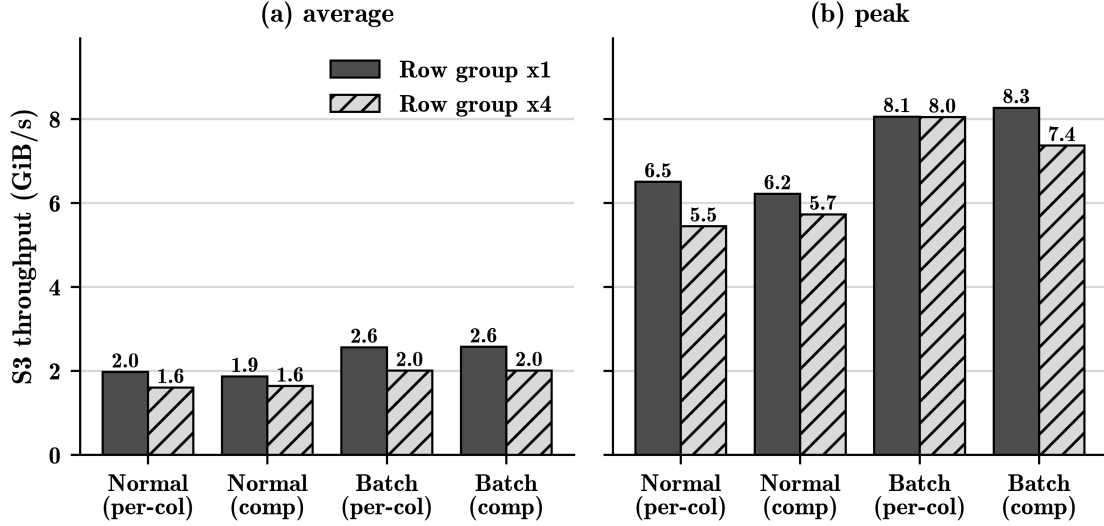


Figure 5.16: TPC-DS S3 throughput across the 99 queries, (a) average and (b) peak, by layout and row group size. Coalescing lifts both: batch prefetching peaks around 8 GiB/s against normal prefetching’s 6 GiB/s, though the average stays near 2.6 GiB/s, as many TPC-DS queries are not bandwidth-bound.

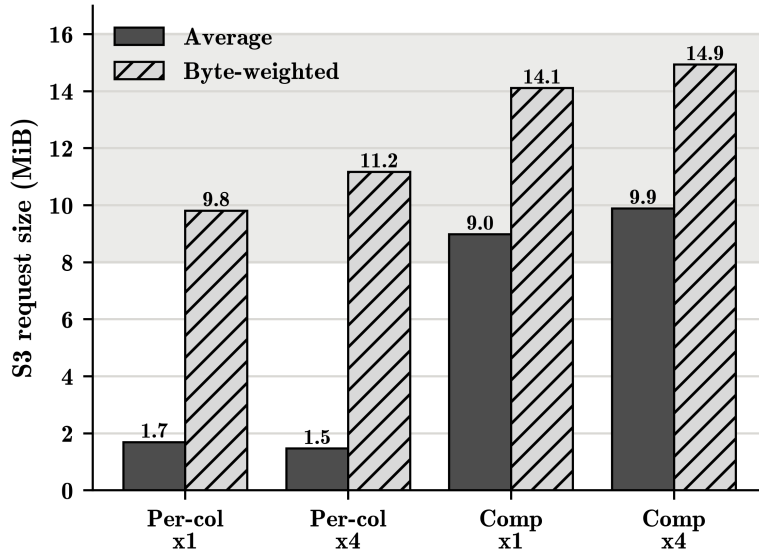


Figure 5.17: Batch prefetching (bp256) S3 request size, simple average versus byte-weighted, by layout and row group size (TPC-DS SF=500). The shaded band is the 8–16 MiB range S3 serves best. Even where the simple average is small (per-column), the byte-weighted size shows that most bytes arrive in large requests.

Cost drops by an order of magnitude, but there is a difference compared to TPC-H.

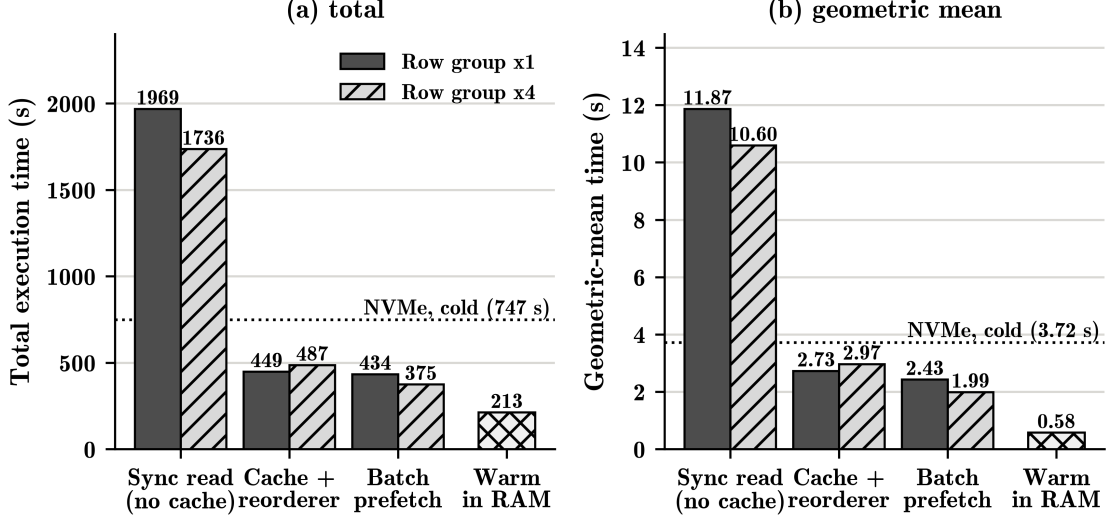


Figure 5.18: TPC-DS whole picture: (a) total and (b) geometric-mean cold time for the best configuration at each stage, rgsx1 vs rgsx4. The dotted line is vanilla DuckDB on local NVMe read cold (747 s); the final bar is the same engine with the data already warm in its buffer pool (213 s total, 0.58 s geometric mean), which does no I/O at all and is therefore the floor a remote read cannot perform better. The naive no-cache stage uses the default single-buffer layout; the two optimized stages use the compacted layout. The vanilla-S3 comparison is reported separately rather than as a stage here.

Because TPC-DS is compute-bound, compute dominates at every configuration: the naive read path costs \$4.23 (\$2.78 compute, \$1.45 requests for its 3.6 million 256 KiB GETs). In contrast, TPC-H was request-dominated. Batch prefetching on compacted still coalesces the request charge away (40× fewer requests, under \$0.04) and, with the 5.3× speedup. The total cost goes down from \$4.23 to \$0.56 (7.5×), as it runs faster.

Compared to TPC-H, TPC-DS is more compute-bound, so I/O optimizations deliver proportionally smaller speedups. Normal prefetching matches rather than beats caching, and compute dominates the cost rather than the S3 request count.

5.5 Summary of the results

Across two benchmarks, TPC-H and TPC-DS, both at a scale factor of 500 and measured cold on a single `m6idn.16xlarge`, the evaluation supports a consistent story. Table 5.4 summarizes the key numbers: total and geometric-mean time and cost per run for the key configurations; the findings below draw on it. We report the geometric mean alongside the total because the total can be dominated by a few long queries, and the two agree on the

5. PERFORMANCE EVALUATION

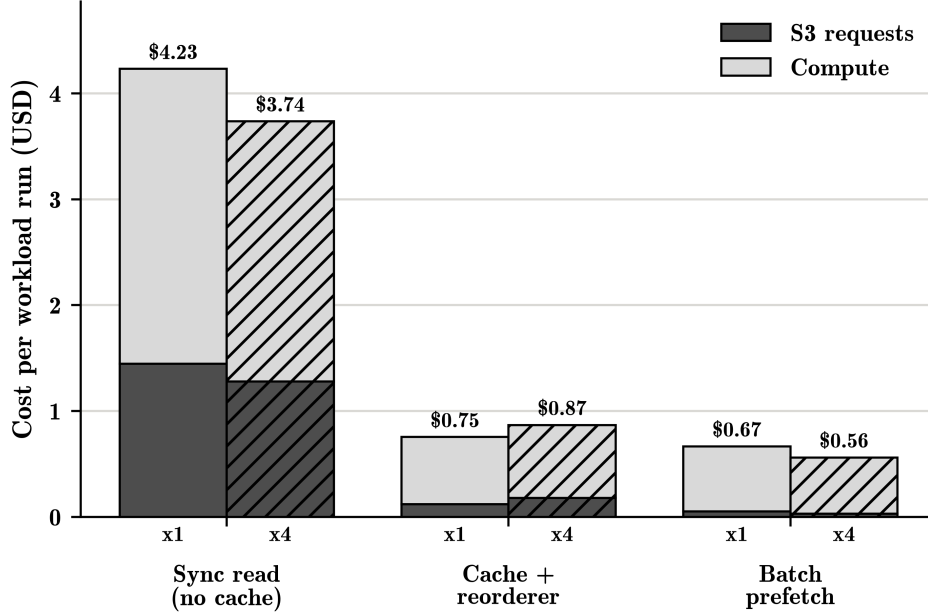


Figure 5.19: TPC-DS cost per workload run across the journey, split into S3-request and compute charges. TPC-DS is compute-bound, so compute is the larger share throughout; the fine-grained synchronous read still carries a large (\$1.45) request charge that coalescing under caching and prefetching removes.

ordering throughout. We present the findings here.

Finding 1: the read path, not S3’s bandwidth, is the bottleneck. A naive synchronous read via differential storage is slow, 1147 s on TPC-H and 1969 s on TPC-DS. This is because DuckDB issues one small 256 KiB request at a time. S3 is not slow, as its aggregate bandwidth is around 11.6 GiB/s. Gains will come from changing how the read path issues requests rather than from S3.

Finding 2: the write layout decides what the read path can do. A single shared write buffer mixes many columns into each 16 MiB object, so a column scan cannot coalesce its reads. Co-locating each column (the per-column writer) and grouping column combinations offline (compaction) make large, useful requests possible. This shows up as an ordering: (1) compacted; (2) per-column; (3) single-buffer. Queries benefit from *semantic* locality, i.e., keeping the columns a query reads in one object, as they can coalesce requests efficiently on physical layout.

5.5 Summary of the results

Benchmark	Configuration	Total (s)	GM (s)	Cost (\$)
TPC-H	Vanilla DuckDB, local NVMe	276	9.30	0.39 [†]
	Vanilla DuckDB, S3 (https)	1666	63.0	2.36 [†]
	Synchronous read, no cache	1147	42.8	2.64
	Cache with reorderer (compacted)	181	7.0	0.31
	Normal prefetch (compacted, x4)	147	5.7	0.95
	Batch prefetch (compacted, x4)	104	4.0	0.17
TPC-DS	Vanilla DuckDB, local NVMe	747	3.72	1.06 [†]
	Vanilla DuckDB, S3 (https)	2533	15.3	3.58 [†]
	Synchronous read, no cache	1969	11.9	4.23
	Cache with reorderer (compacted)	449	2.73	0.75
	Normal prefetch (compacted, x4)	462	2.55	1.90
	Batch prefetch (compacted, x4)	375	1.99	0.56

Table 5.4: Summary across configurations (SF=500, cold): total time, geometric-mean per-query time, and cost per workload run. The best configuration is in bold. Caching and synchronous rows are at the default row group size (rgsx1); the prefetch rows are at rgxsx4. [†]Compute only: local disk issues no S3 requests, and native DuckDB’s S3-request count is not instrumented, so no request charge is included for the vanilla rows.

Finding 3: caching is worthless without the reorderer. Because DuckDB hands out row groups sequentially and consecutive row groups sit close together, all threads hit the same object. Optimistic caching alone therefore barely moves the baseline on TPC-H (from 1147 to 905–1024s) and performs worse on TPC-DS (from 1969 to 2154–2374s). The modulo reorderer breaks the sequential hand-out so that cached objects are reused a few steps later; only then does caching pay off, increasing the hit rate on the clean layouts from 29–33% to 71–90% on TPC-H and 93–95% on TPC-DS, and collapsing the time by 5.7× on TPC-H’s compacted layout and 5.2× on TPC-DS’s. Reuse and usefulness are orthogonal: the reorderer supplies reuse, the layout supplies usefulness, and both are needed.

Finding 4: cloud object storage can beat cloud-attached local SSDs. Both routes to concurrency are faster than local disk, but not equally robustly. Caching is faster only once the layout is clean: with the reorderer on the *compacted* layout it reaches 181s against vanilla DuckDB’s 275s on a local cloud-based NVMe SSD on TPC-H, and 449s against 747s on TPC-DS. On the per-column layout it is slower, 455s against the same 275s, because at a 71% hit rate roughly one read in three still goes to S3. Prefetching is

5. PERFORMANCE EVALUATION

faster everywhere: all prefetching configurations we measured beat local disk, 103–185 s on TPC-H and 375–599 s on TPC-DS, including all six single-buffer configurations on TPC-DS. The layout still decides by how much prefetching wins. One reason covers both routes: aggregate S3 bandwidth (11.6 GiB/s) exceeds that of a single NVMe SSD (2.5 GiB/s).

Finding 5: prefetching with coalescing is the winner, though mostly on cost.

Adding the prefetching onto the read path and coalescing a batch of upcoming row groups into a few large GETs is the fastest configuration on both benchmarks. Batch prefetching merges 24–36 blocks per request on the compacted layout, cutting the number of S3 fetches from millions to tens of thousands. Compared to the synchronous no-cache baseline, the request count drops 47.9 $\times$ on TPC-H (2.45 million to 51,197) and 46.3 $\times$ on TPC-DS (3.62 million to 78,048). The best configuration reaches 104 s on TPC-H (16 $\times$ faster than vanilla S3, 2.6 $\times$ faster than local NVMe) and 375 s on TPC-DS (2.0 $\times$ faster than local NVMe). However, its execution-time gain over normal prefetching is modest, only 20–30%; coalescing’s main contribution is cost, cutting requests by 24–40 $\times$ and cost by 3–6 $\times$ (Finding 8).

Finding 6: caching and prefetching are two ways to better execution time, with different trade-offs.

Both hide S3 latency behind concurrency and can in principle reach a similar throughput: at a perfect hit rate, caching turns each 256 KiB read into a full 16 MiB of useful data. Caching is the simpler mechanism. It sits entirely in the differential-storage layer and needs no changes to DuckDB. However, it is **indirect**: it is helpful only with a clean layout and the reorderer, is sensitive to data locality and row group order, and gives no direct control over how many requests are in flight. Prefetching is more invasive, hooking the engine’s scan through the reorderer. Still, far more controllable: it knows exactly which row groups the scan will touch and sets the number of concurrent requests and how aggressively to coalesce them. Caching is the cheaper mechanism to build; prefetching is the one that lets us control concurrency directly.

Finding 7: a larger row group is a cheap and useful knob.

Quadrupling the row group size (rgsx4) cuts multi-segment blocks from 36.7% to 6.4%, so a column scan reads fewer wasted bytes (1.32 $\times$ less on TPC-H) and coalesces better. It is faster and cheaper with every configuration on TPC-H, and with synchronous-read and prefetching on TPC-DS; the one exception is caching with the reorderer on TPC-DS, which it is not better.

Finding 8: cost is driven by request granularity, not just runtime. Under the cost model of Equation (5.1), normal access incurs a large per-request charge on top of compute: normal prefetching on TPC-H pays \$0.98 of its \$1.21 cost in S3 requests, and even TPC-DS’s naive read carries a \$1.45 request charge. Coalescing removes almost all of it, cutting the total cost 7–15 $\times$. The cheapest configuration is therefore also among the fastest.

In short, with a column-aware write layout, a strided reorderer, and coalesced prefetching, DuckDB executing against S3 is not only viable but faster and cheaper than executing against cloud-based local disk, on both benchmarks.

5. PERFORMANCE EVALUATION

6

Discussion

6.1 Reflecting on research questions and related work

6.1.1 Research questions

RQ1: What is the current access pattern of DuckDB with differential storage, and what is the S3 throughput we can currently achieve?

We answered RQ1 by characterizing DuckDB’s access patterns. On the write path, we looked into how DuckDB writes data and handles 256 KiB blocks with the `PartialBlockManager`. We explained the phenomenon of *multi-segment blocks*, which reduce the chance of coalescing. On the S3 layout with differential storage, we found that the single-buffer writer packs many columns into one 16 MiB object (Figure 4.5). This means blocks containing data from the same column stay in different locations, so we cannot merge them into large S3 requests. We differentiate between the logical and physical positions by measuring both the logical reads DuckDB issues and the physical GETs that actually reach S3 on TPC-H and TPC-DS, showing how blocks and row groups are distributed on S3. A MotherDuck workload is analyzed to give hints into how the data is ingested and laid out (Section 4.2.3). Finally, we showed the throughput baseline that the rest of the thesis works on: fetching 256 KiB blocks on demand, one per request, is latency-bound and sustains only a few hundred MiB/s, orders of magnitude smaller than the network capability of the instance and the available bandwidth of S3 (Chapter 5, Finding 1). RQ1 concludes that S3’s bandwidth does not limit the storage layer; rather, the limits are how the read path issues requests and a write layout that makes coalescing hard.

RQ2: What changes could one make to differential storage’s write path to improve S3 throughput?

6. DISCUSSION

RQ2 showed how the write path could be changed to support larger S3 requests. We pointed out that the mixed-column layout is an obstacle to coalescing, and routing each column’s blocks to its own object, the *per-column* layout (Figure 4.7), is the first approach towards *semantic* locality: the blocks a scan needs are placed together. Hence, a single large GET fetches data the query actually uses rather than a mixture of irrelevant column data (Chapter 5, Finding 2). Multi-segment blocks (Section 4.2.2) still fragment these runs, so we mitigated them from the write side by using a larger row group, which reduces the number of multi-segment blocks (Finding 7), and, for the cleanest possible layout, we added an offline *compaction* pass that sorts blocks into per-column-combination objects (Algorithm 1). However, the write path alone is not enough. DuckDB still issues 256 KiB synchronous reads, which does not turn that locality into larger requests (Finding 2). Improving the layout therefore only shows its effects once we improve the read paths, which is exactly what motivated RQ3.

RQ3: What changes could one make in DuckDB’s read path to improve S3 throughput, and how do they interact with the layout from RQ2?

RQ3 addressed the read path and its interaction with the layout. Because DuckDB has no asynchronous I/O yet¹ and it limits concurrency to its thread count; the naive path cannot keep enough requests in flight to saturate a fast network link. We explored two read-path mechanisms on top of the per-column layout. The first is *caching*: any block read optimistically pulls its whole 16 MiB object into a local disk cache. On its own, this barely helped, because DuckDB hands out row groups sequentially and all threads contend for the same object at once; improving reuse with the *reorderer* (Algorithm 2), a hook we repurposed from DuckDB’s row group reordering, made caching effective (Chapter 5, Finding 3) and reached performance comparable to prefetching (Finding 6). The second is *prefetching*: using the same reorderer hook to fetch upcoming row groups ahead of the scan (Algorithm 3), with *batch* prefetching to coalesce their blocks into large, position-sorted GETs. This gave the best overall performance and, by merging millions of small requests into a few thousand large ones, cut costs significantly (Finding 5, Finding 8). Both mechanisms confirm the interaction we expected (Figure 5.18): prefetching removes the concurrency bottleneck. Still, the per-column and compacted layout of RQ2 enables

¹DuckDB just announced a feature named asynchronous I/O (<https://duckdb.org/2026/07/31/asynchronous-io>). Fetches for upcoming work are scheduled early onto a pool of threads that perform *blocking* reads, and a worker that reaches a job whose data has not arrived still parks its scan task. It also combines nearby byte ranges, making it the same class of mechanism as the batch prefetching of Chapter 4. Two things keep it separate from this work: it covers Parquet and CSV, with DuckDB’s native format listed as still to come.

us to issue the large, useful requests that actually saturate S3. Caching and prefetching are thus two ways to improve throughput, with opposite trade-offs (Finding 6): caching is simpler because it lives entirely in the differential-storage layer, but it is indirect and sensitive to layout and row group order, whereas prefetching is more invasive but provides direct control over request size and concurrency.

6.1.2 Related work

The systems closest to this work are AnyBlob [27] and Pixels [18], which also target high-throughput analytics over cloud object storage. Our advantage is on the write and layout side: because differential storage sits between DuckDB and S3 and addresses data in small, independent 256 KiB blocks, it can rearrange those blocks essentially for free, without touching DuckDB’s on-disk format. We are therefore free to lay out data however best serves reading, in particular by grouping a column’s blocks so that a few large requests serve a scan. AnyBlob and Pixels have far less freedom here, since they read an existing on-storage format rather than owning the placement.

Our weakness is on the read path. Both AnyBlob and Pixels are built with asynchronous I/O, which lets them control concurrency and compute resources: they keep many requests in flight, overlap them with compute, and throttle them adaptively. DuckDB has no such mechanism yet, so we increase concurrency through prefetching or caching.

6.2 Future work

Adaptive S3 traffic per query. Our prefetcher issues S3 requests aggressively, on the assumption that there is spare resource capacity to hide latency. For a compute-bound query, this is wasteful: since the CPU is the bottleneck, prefetching adds bandwidth and cost, because the query cannot consume the data any faster than it can compute on it. Systems such as AnyBlob and Pixels instead adapt a query’s S3 traffic to how it is using the CPU, throttling request concurrency while the query is compute-bound and opening it up when it stalls on I/O. The decision can be driven by runtime signals from within the query, for example whether worker threads are blocked waiting on I/O or busy on compute. Building such a scheme on top of our prefetcher, so that a single query backs off S3 traffic while it is compute-bound and ramps up only when I/O sits on its critical path, is a natural next step.

6. DISCUSSION

Benchmarking more complex write patterns. Our layouts are all produced by a single bulk ingest, and we evaluate reads over that static layout. Real workloads can vary. For example, a real-world workload could involve incremental modification: streams of small inserts, deletions, and updates. Small inserts might produce more partially filled blocks than the long, clean runs a bulk load gives; deletions leave holes which fragment objects; and updates rewrite blocks out of order, all of which can affect the semantic locality the read path depends on. Evaluating the per-column and compacted layouts and studying how the coalescing efficiency goes under such workloads would show whether the gains shown here survive a realistic workload.

Pushing semantic locality further. Our layouts group blocks by the column they belong to, or, after compaction, by the full set of columns a block contains. Both keys are properties of the data, not of the workload: they are the best we can do knowing nothing about how the table will be queried. But the real strength of differential storage is that it is free to reorder blocks on S3 however we like, so nothing forces the grouping key to be the column. An approach is a workload-aware layout, in which the key is obtained from observed query behaviors: columns that are regularly accessed together are placed in the same objects, so that a single large GET combines several of the columns a query needs, rather than one column at a time. This matches what we already measured, since queries touch only a small fraction of a table’s columns (Section 4.2.3) and those column sets recur throughout the workload, making co-access predictable from query history. Pixels does something adjacent when it orders column chunks within a row group by co-access, solving the placement problem with simulated annealing [18; 19].

Asynchronous I/O. Another approach to this problem is asynchronous I/O in DuckDB. Prefetching and caching have limitations. Because the engine cannot overlap I/O with compute within a query, we hide latency by issuing reads ahead of time, which implicitly assumes that there is spare capacity to do that extra work. Asynchronous I/O would remove that assumption. Instead of speculatively fetching, a thread that stalls on a read could continue working on other tasks in the same query; however, whether we can issue bigger requests still relies on a layout that facilitates it.

7

Conclusion

In this thesis, we showed different ways to improve the performance of executing DuckDB’s single-file storage format on cloud object storage using differential storage. DuckDB is designed for fast local disk access, reading 256 KiB blocks synchronously. However, cloud object storage runs under different requirements: files are immutable, requests have tens of milliseconds of latency, and network bandwidth is achieved only with requests of 8–16 MiB and with hundreds of them concurrently. MotherDuck’s differential storage solved the immutability problem by maintaining a mapping from logical positions of the database file to physical positions on S3. This thesis investigated how to close the performance gap and showed that the answer lies in the block layout on cloud object stores and how the read path requests them.

We began by analyzing the system’s current behavior. Our baseline experiments on S3 confirmed previous findings: using 256 KiB requests, we can only saturate one-third the throughput that we can achieve using 8–16 MiB requests. At the same time, we also have to maintain significantly higher concurrency. Analyzing DuckDB’s data blocks on S3 showed why the default writer cannot enable us to achieve higher request sizes. Since differential storage appends each flushed block to a shared buffer, a 16 MiB object typically contains a median of 16 distinct columns on TPC-H and 24 on TPC-DS. Consequently, scans involving a subset of columns can not do coalescing effectively. Additionally, we identified a challenge: multi-segment blocks, which make coalescing harder.

Based on this analysis, we proposed changes to both the read path and write path. The first is to **increase request granularity**, which is a write-path concern: we route each block to a per-column buffer so that a column’s blocks share the same objects, increase the row group size so that most column–row-groups fill a whole block, and add an offline compaction pass that regroups the multi-segment blocks into per-column-combination

7. CONCLUSION

objects. All of these are to create a cleaner layout for caching and prefetching. The second is to **increase concurrency**, a read-path concern, which we explored in two forms. Optimistic caching turns every 256 KiB read into a 16 MiB object fetch held in a local disk cache; reorderer-based prefetching instead uses DuckDB’s row-group reordering hook to fetch upcoming row groups ahead of the scan, optionally coalescing a whole batch of their blocks into a few large GETs.

We provided concrete results for each approach. On TPC-H at SF=500, vanilla DuckDB reading its database file from S3 takes 1666s; routing reads through differential storage brings this to 1147s. Caching is inefficient on its own, because DuckDB hands out row groups sequentially and all threads consume the same object at once. Combining a clean layout and the modulo reorderer increases the cache hit rate from around 30% to 90%. It achieves a total execution time of 181s, faster than vanilla DuckDB on a local cloud-based NVMe SSD (275s). Batch prefetching on the compacted layout with a larger row group is the best configuration we saw, at 104s: about $16\times$ faster than vanilla DuckDB on S3 and $2.6\times$ faster than the same engine on local disk. Benchmarking on TPC-DS showed the same result: from 2533s for vanilla S3 and 1969s for the naive read, we reach 449s using caching and 375s using batch prefetching, $2.0\times$ faster than local NVMe. Both best configurations are only about $1.8\times$ slower than the same engine running entirely in memory: 104 against 61s on TPC-H, and 375 against 213s on TPC-DS.

In a cloud environment, cost is as important as query execution time. Cloud providers charge based on request counts rather than bytes transferred. We can merge 24–40 blocks on average into one request on the clean layout, so the request count decreases from millions of 256 KiB GETs to tens of thousands. That reduces almost all of the request costs, and because larger requests also maintain higher throughput, it also makes the run faster and, with it, the compute charge. The cost of a TPC-H run drops from \$2.64 for the fine-grained synchronous read to \$0.17, about $15\times$, and a TPC-DS run from \$4.23 to \$0.56. The cheapest configuration we measured is also among the fastest, which is a practical argument for making requests bigger.

Regarding the three research questions, RQ1 showed that S3’s bandwidth is not the limit; the read path and the layout are. RQ2 showed that the write path can produce the *semantic* locality a query needs by keeping the columns a scan reads in the same object. Yet that layout alone cannot do better, as the read path still reads one block at a time. RQ3 closed the gap: prefetching removes the concurrency bottleneck, yet only the per-column and compacted layouts in RQ2 allow it to issue both large and useful requests. The two parts are inseparable.

In comparison to related work, the primary advantage of differential storage is its flexibility. By mediating DuckDB’s file at the level of small, independent 256 KiB blocks, it enables block rearrangement without modifying DuckDB’s on-disk format, allowing data to be organized optimally for reading. AnyBlob [27] determines block placement at write time, aligning column chunks with S3’s preferred request ranges, but lacks mechanisms for subsequent data reorganization. Pixels [18] is more closely aligned with our approach: it reorders column chunks within a row group. It compacts multiple row groups, placing column data in order, copying bytes without decoding, similar to our compaction process. However, Pixels [18] requires changing the metadata, whereas our method is format-agnostic; blocks remain unchanged as written by DuckDB, with only the mapping updated. This indirection allows the addition of a per-column writer and offline compaction to a general-purpose engine without changing its storage format. Both AnyBlob [27] and Pixels [18] use asynchronous I/O, treating concurrency as a primary scheduling concern. DuckDB has no such mechanism yet, so our caching and prefetching strategies restore concurrency indirectly by prefetching ahead, assuming sufficient memory and bandwidth. In practice, these methods are effective. An alternative solution would involve implementing asynchronous I/O within DuckDB, enabling threads stalled on reads to yield and progress elsewhere in the same query, thereby overlapping I/O and computation by design. However, in short, a column-aware write layout, strided reordering, and coalesced prefetching suffice to make DuckDB on cloud object storage not only viable but also faster and more cost-effective than DuckDB on cloud-based local disk.

7. CONCLUSION

References

- [1] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. “Data page layouts for relational databases on deep memory hierarchies.” In: *The VLDB Journal* 11.3 (2002), pp. 198–215. URL: <https://doi.org/10.1007/s00778-002-0074-9> (8, 9).
- [2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. “Weaving Relations for Cache Performance.” In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB ’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 169–180 (9, 29).
- [3] Amazon Web Services. *Amazon EBS Volume Types*. <https://aws.amazon.com/ebs/volume-types/>. Accessed: 2026-07-14. 2026 (32).
- [4] Amazon Web Services. *Amazon EC2 Instance Type Explorer*. <https://aws.amazon.com/ec2/instance-explorer/>. Accessed: 2026-07-12. 2026 (19, 32).
- [5] Amazon Web Services. *Amazon EC2 M6i Instances*. <https://aws.amazon.com/ec2/instance-types/m6i/>. Accessed: 2026-07-14. 2026 (38).
- [6] Amazon Web Services. *Amazon EC2 On-Demand Pricing*. <https://aws.amazon.com/ec2/pricing/on-demand/>. Accessed: 2026-07-12. 2026 (20).
- [7] Amazon Web Services. *Amazon Elastic Block Store*. <https://aws.amazon.com/ebs/>. Accessed: 2026-07-14. 2026 (1, 31, 32).
- [8] Amazon Web Services. *Amazon Elastic File System (EFS)*. <https://aws.amazon.com/efs/>. Accessed: 2026-07-27. 2026 (1).
- [9] Amazon Web Services. *Amazon S3 Storage Classes*. <https://aws.amazon.com/s3/storage-classes/>. Accessed: 2026-12-07. 2026 (18).
- [10] Amazon Web Services. *AWS Common Runtime (CRT) Libraries*. <https://docs.aws.amazon.com/sdkref/latest/guide/common-runtime.html>. Accessed: 2026-07-14. 2026 (37, 65).
- [11] Amazon Web Services. *Performance Guidelines for Amazon S3*. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance-guidelines.html>. Accessed: 2026-07-14. 2026 (38, 39).

REFERENCES

- [12] Amazon Web Services. *Specifications for Amazon EC2 General Purpose Instances*. <https://docs.aws.amazon.com/ec2/latest/instancetypes/gp.html>. Accessed: 2026-07-14. 2026 (32).
- [13] Apache Software Foundation. *Apache ORC: Indexes*. <https://orc.apache.org/docs/indexes.html>. Accessed: 2026-07-13 (27).
- [14] Apache Software Foundation. *Parquet File Format: PageIndex*. <https://parquet.apache.org/docs/file-format/pageindex/>. Accessed: 2026-07-13 (27, 28).
- [15] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja uszczak, Michal witakowski, Micha Szafranski, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. “Delta lake: high-performance ACID table storage over cloud object stores.” In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 3411–3424. URL: <https://doi.org/10.14778/3415478.3415560> (19, 21).
- [16] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinsky, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. “Amazon Redshift Re-invented.” In: *Proceedings of the 2022 International Conference on Management of Data. SIGMOD ’22*. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 2205–2217. URL: <https://doi.org/10.1145/3514221.3526045> (1, 18).
- [17] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. “Photon: A Fast Query Engine for Lakehouse Systems.” In: *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*. ACM, 2022, pp. 2326–2339 (18).
- [18] Haoqiong Bian and Anastasia Ailamaki. “Pixels: An Efficient Column Store for Cloud Data Lakes.” In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2022, pp. 3078–3090 (4, 28–31, 35, 36, 38, 99, 100, 103).
- [19] Haoqiong Bian, Ying Yan, Wenbo Tao, Liang Jeff Chen, Yueguo Chen, Xiaoyong Du, and Thomas Moscibroda. “Wide Table Layout Optimization based on Column Ordering and Duplication.” In: *Proceedings of the 2017 ACM International Conference on Management of Data. SIGMOD ’17*. Chicago, Illinois, USA: Association for

REFERENCES

- Computing Machinery, 2017, pp. 299–314. URL: <https://doi.org/10.1145/3035918.3035930> (30, 100).
- [20] Al Brown. *Top 5 Cloud Data Warehouses in 2026: Architecture, Cost, and Open-Source*. ClickHouse. Last updated April 13, 2026; accessed July 6, 2026. Apr. 2026. URL: <https://clickhouse.com/resources/engineering/top-5-cloud-data-warehouses> (1).
- [21] George P. Copeland and Setrag N. Khoshafian. “A decomposition storage model.” In: *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’85. Austin, Texas, USA: Association for Computing Machinery, 1985, pp. 268–279. URL: <https://doi.org/10.1145/318898.318923> (9).
- [22] Andrew Crotty, Viktor Leis, and Andrew Pavlo. “Are You Sure You Want to Use MMAP in Your Database Management System?” In: *CIDR 2022, Conference on Innovative Data Systems Research*. 2022 (7).
- [23] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. “The Snowflake Elastic Data Warehouse.” In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 215–226. URL: <https://doi.org/10.1145/2882903.2903741> (1, 18).
- [24] Jeffrey Dean and Luiz André Barroso. “The tail at scale.” In: *Commun. ACM* 56.2 (Feb. 2013), pp. 74–80. URL: <https://doi.org/10.1145/2408776.2408794> (19).
- [25] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. “RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications.” In: *ACM Trans. Storage* 17.4 (Oct. 2021). URL: <https://doi.org/10.1145/3483840> (7).
- [26] DuckDB. *Storage Versions and Format — DuckDB Documentation*. <https://duckdb.org/docs/current/internals/storage>. Accessed: 2026-07-06. 2026 (7, 9).
- [27] Dominik Durner, Viktor Leis, and Thomas Neumann. “Exploiting Cloud Object Storage for High-Performance Analytics.” In: *Proc. VLDB Endow.* 16.11 (July 2023), pp. 2769–2782. URL: <https://doi.org/10.14778/3611479.3611486> (4, 6, 18–20, 27, 30–38, 71, 99, 103).

REFERENCES

- [28] Gabriel Haas and Viktor Leis. “What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines.” In: *Proc. VLDB Endow.* 16.9 (May 2023), pp. 2090–2102. URL: <https://doi.org/10.14778/3598581.3598584> (5, 6).
- [29] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2017. URL: <https://books.google.nl/books?id=MBQFuAEACAAJ> (5, 6).
- [30] Joseph Hwang. *Differential Storage: A Key Building Block for a DuckDB-Based Data Warehouse*. MotherDuck Blog. Accessed: 2026-07-06. Mar. 2024. URL: <https://motherduck.com/blog/differential-storage-building-block-for-data-warehouse/> (1, 20, 22).
- [31] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. “Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation.” In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 311–326. URL: <https://doi.org/10.1145/2882903.2882925> (35).
- [32] Viktor Leis. *SSDs Have Become Ridiculously Fast, Except in the Cloud*. Database Architects (blog). Accessed: 2026-07-28. Feb. 2024. URL: <http://databasearchitects.blogspot.com/2024/02/ssds-have-become-ridiculously-fast.html> (70).
- [33] Seth Markle and James Bornholt. *AWS re:Invent 2024 — Dive deep on Amazon S3 (STG302)*. Timestamp 5:52. Amazon Web Services. Dec. 5, 2024. URL: <https://www.youtube.com/watch?v=NXehLy7IiPM&t=352s> (visited on 07/27/2026) (18).
- [34] Microsoft. *Azure Disk Storage*. <https://azure.microsoft.com/en-us/products/storage/disks>. Accessed: 2026-07-27. 2026 (1).
- [35] Microsoft. *Azure Files*. <https://azure.microsoft.com/en-us/products/storage/files>. Accessed: 2026-07-27. 2026 (1).
- [36] Thomas Neumann and Michael J. Freitag. “Umbral: A Disk-Based System with In-Memory Performance.” In: *Conference on Innovative Data Systems Research*. 2020. URL: <https://api.semanticscholar.org/CorpusID:209379505> (7).
- [37] Andy Pavlo. *Lecture #03: Database Storage (Files & Pages), 15-445/645 Database Systems*. Carnegie Mellon University, <https://15445.courses.cs.cmu.edu/spring2026/>. Accessed: 2026-07-06. 2026 (7).

REFERENCES

- [38] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database.” In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1981–1984. URL: <https://doi.org/10.1145/3299869.3320212> (9, 20, 26).
- [39] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. “ClickHouse - Lightning Fast Analytics for Everyone.” In: *Proc. VLDB Endow.* 17.12 (Aug. 2024), pp. 3731–3744. URL: <https://doi.org/10.14778/3685800.3685802> (1, 26).
- [40] The PostgreSQL Global Development Group. *Database File Layout — PostgreSQL Documentation*. <https://www.postgresql.org/docs/current/storage-file-layout.html>. Accessed: 2026-07-06 (7).
- [41] Jordan Tigani and Boaz Leskes. *Separating Storage and Compute in DuckDB*. MotherDuck Blog. Accessed: 2026-07-06. Dec. 2024. URL: <https://motherduck.com/blog/separating-storage-compute-duckdb/> (1).
- [42] Trino Software Foundation. *Trino: A Fast Distributed SQL Query Engine for Big Data Analytics*. <https://trino.io/>. Accessed: 2026-07-13. 2026 (26, 27).
- [43] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. “Apache Spark: a unified engine for big data processing.” In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. URL: <https://doi.org/10.1145/2934664> (27).
- [44] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. “An Empirical Evaluation of Columnar Storage Formats.” In: *Proc. VLDB Endow.* 17.2 (Oct. 2023), pp. 148–161. URL: <https://doi.org/10.14778/3626292.3626298> (9).
- [45] Xinyu Zeng, Ruijun Meng, Martin Prammer, Wes McKinney, Jignesh M. Patel, Andrew Pavlo, and Huanchen Zhang. “F3: The Open-Source Data File Format for the Future.” In: *Proc. ACM Manag. Data* 3.4 (Sept. 2025). URL: <https://doi.org/10.1145/3749163> (25, 27, 28, 36).