Fusion: An Analytics Object Store Optimized for Query Pushdown

Jianan Lu*
Princeton University
Princeton, United States
jiananl@princeton.edu

Ashwini Raina*
Princeton University
Princeton, United States
araina@princeton.edu

Asaf Cidon
Columbia University
New York, United States
asaf.cidon@columbia.edu

Michael J. Freedman
Princeton University
Princeton, United States
mfreed@cs.princeton.edu

Abstract

The prevalence of disaggregated storage in public clouds has led to increased latency in modern OLAP cloud databases, particularly when handling ad-hoc and highly-selective queries on large objects. To address this, cloud databases have adopted computation pushdown, executing query predicates closer to the storage layer. However, existing pushdown solutions are inefficient in erasure-coded storage. Cloud storage employs erasure coding that partitions analytics file objects into fixed-sized blocks and distributes them across storage nodes. Consequently, when a specific part of the object is queried, the storage system must reassemble the object across nodes, incurring significant network latency.

In this work, we present *Fusion*, an object store for analytics that is optimized for query pushdown on erasure-coded data. It co-designs its erasure coding and file placement topologies, taking into account popular analytics file formats (e.g., Parquet). Fusion employs a novel stripe construction algorithm that prevents fragmentation of computable units within an object, and minimizes storage overhead during erasure coding. Compared to existing erasure-coded stores, Fusion improves median and tail latency by 64% and 81%, respectively, on TPC-H, and up to 40% and 48% respectively, on real-world SQL queries. Fusion achieves this while incurring a modest 1.2% storage overhead compared to the optimal.

CCS Concepts: • **Information systems → Cloud based storage**.

Keywords: data analytics; erasure codes; distributed storage

ACM Reference Format:
Jianan Lu, Ashwini Raina, Asaf Cidon, and Michael J. Freedman. 2025. Fusion: An Analytics Object Store Optimized for Query Pushdown. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3669940.3707234

*Both authors contributed equally to this research.

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707234

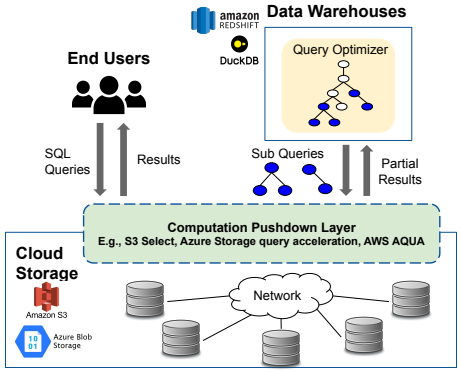


Figure 1. Existing architecture of computation pushdown within the domain of big data analytics in the cloud.

1 Introduction

Major clouds operators like AWS, Azure, and GCP have embraced disaggregated storage architectures. Under such paradigms, raw data is uploaded to a highly-available object store such as AWS S3 or Azure Storage. When needed, data is fetched into a compute cluster to run compute-intensive jobs. While storage disaggregation helps improve resource utilization and elasticity in public clouds, it can hurt the performance of ad-hoc, interactive analytics queries on large objects. Prior work [47, 93, 99] shows that object stores are dominated by large objects and large object reads are prevalent. In one storage production trace from Microsoft [47], over 60% of objects have size greater than 1GB. Meanwhile, real-world workloads are highly selective. BigQuery [59, 85] reports that about 50% of queries return less than 1% of the data. Transferring a large object over the network while eventually using only a small portion of its data not only wastes valuable network bandwidth but also significantly degrades query performance.

To improve the performance of ad-hoc, interactive queries, analytics systems deploy a classic technique called *computation pushdown*, which ships compute closer to storage to avoid unnecessary data movement. Figure 1 depicts the high-level architecture of computation pushdown for big data analytics in the cloud. Users can submit simple SQL-like queries to be executed closer to the object stores. Upstream database query engines can also push down parts of query plans to the object stores, such as highly-selective predicates and aggregates, thereby reducing the data volume sent to the

data warehouse. In recent years, computation pushdown is embraced by data warehousing platforms [5, 55, 72, 105, 109] and storage systems [7, 11, 16].

Despite its popularity, we argue that existing computation pushdown solutions on cloud object stores are inefficient in the very common scenario of (a) erasure-coded data, and (b) data stored in analytics file formats. First, most cloud object stores [1, 35, 47, 76, 87, 93, 103] use erasure coding due to its significantly lower cost compared to replication. Objects are striped into fixed-sized data and parity blocks and distributed across multiple storage nodes for fault tolerance. Data placement and recovery are done at the granularity of these data and parity blocks. Second, popular analytics file formats [3, 4, 10, 39] allow queries to execute *only on the relevant parts of a file* without having to read the entire file. For example, Apache Parquet [4] partitions data into a collection of row groups and then each row group is partitioned column-wise into *column chunks*. A column chunk comprises values from a single attribute and directly supports common analytical operations, such as filters, projections, and aggregations. We refer to the granularity of a data unit capable of executing a part of the query (e.g., filter operations) on its own as the *smallest computable unit*.

Since erasure codes treat data as a blob of bytes, they are unaware of the internal semantics of the highly structured objects. Moreover, the smallest computable units tend to have variable sizes influenced by several factors, such as the data type, cardinality and compression. Today, when an analytics file object is erasure coded into fixed-sized blocks across storage nodes, each of its smallest computable units may be split across multiple nodes. Consequently, existing object stores such as MinIO [35] and Ceph [103] must first reassemble these smallest computable units across storage nodes before executing a query. This wastes precious network bandwidth and network processing CPU in the storage cluster and also incurs extra network latency. The root cause of this sub-optimality is that erasure coding and computation pushdown in modern cloud storage operate in distinct layers of the storage stack, functioning within isolated silos.

We present Fusion, a new object store for analytics that is optimized for computation pushdown. It is designed to efficiently handle *ad-hoc, interactive* analytics queries against object-based storage, akin to popular analytics systems like Amazon Athena [32] and BigQuery [34]. Fusion co-designs the erasure coding topology to take into account popular analytics file formats. One of our main contributions is a novel technique called *file-format-aware coding* (FAC), which prevents the splitting of the smallest computable units across nodes during erasure coding. Prior work [36] pads the original objects to enforce the alignment of these computable units with storage blocks, sharing a similar goal of preventing data splits. However, padding incurs a significant storage overhead, negating the cost advantages of erasure coding.

FAC employs an intelligent *stripe construction algorithm* that departs from the conventional practice of using a fixed block size per stripe [76, 91, 94]. Instead, it uses variable block sizes and stripe sizes to align erasure code blocks seamlessly with the boundaries of the smallest computable units. This creates a new challenge, as naively utilizing erasure coding with variable-sized data blocks dramatically increases storage overhead. To solve this storage overhead challenge, the stripe construction algorithm efficiently bin-packs the smallest computable units, creating data blocks of similar sizes within a stripe, and dynamically adjusts stripe sizes based on the size distribution of the smallest computable units. FAC then utilizes this layout to erasure code the object and store each smallest computable unit *intact* among nodes.

Fusion uses a *fine-grained pushdown* mechanism to optimize query pushdown on FAC-encoded objects. Modern analytics file formats employ aggressive compression and encoding techniques on the smallest computable units to reduce storage costs. For instance, the smallest computable units in the TPC-H lineitem Parquet object can have compression ratios of up to 63. For queries involving such highly compressed smallest computable units, the pushdown operation, after decoding these smallest computable units locally on the storage node, may transfer a substantial portion of the uncompressed data as the query result over the network. This leads to higher network latency and degrades query performance. In such scenarios, it is advantageous to transfer the smallest computable unit over the network in its compressed form rather than processing it in-situ on the storage node. We find that pushdown performance is directly influenced by both the query's selectivity and the compressibility of the smallest computable units it operates on. We design a cost model to capture this relationship and Fusion uses it to determine which smallest computable units benefit the most from query pushdown and enables pushdown only for those smallest computable units.

In summary, this paper makes the following contributions:

1. Novel coding technique (FAC) that co-designs erasure coding with analytics file format to maximize pushdowns.
2. Bin-packing algorithm for variable-sized blocks that minimizes storage overhead and prevents fragmentation of the smallest computable units.
3. A fine-grained, cost-based pushdown mechanism that maximizes pushdown performance.
4. Fusion, a new analytics object store, improves tail latency by up to 81% on TPC-H and 48% on real-world SQL queries, with only a 1.2% storage overhead w.r.t. the optimal.

2 Background

Computation pushdown. Computation pushdown is a decades-old technique where parts of a query are "pushed down" closer to where the data is stored. It is prevalent in

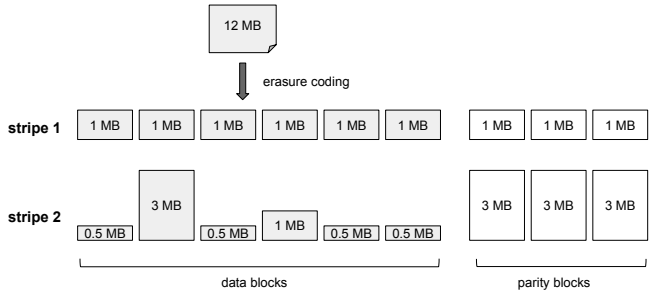


Figure 2. A $(9, 6)$ erasure code partitions a 12MB object into two 6MB data stripes, each consisting of 6 data blocks and 3 parity blocks. Stripe 1 uses a fixed block size of 1MB. Stripe 2 uses variable data block sizes.

big data analytics systems that push down segments of SQL queries, such as filters (e.g., , WHERE clauses) and aggregations (e.g., SUM, AVG), to the storage layer. Some notable examples include AWS S3 Select [7] and Azure Data Lake Storage query acceleration [5]. Computation pushdown significantly improves query latency by reducing the amount of data transferred over the network, thereby saving both network bandwidth and CPU resources for users.

Erasure-coded object stores. Distributed storage systems [1, 35, 47, 62, 73, 76, 87, 92, 93, 103, 104] widely adopt *systematic* erasure codes to tolerate node failures with lower storage overhead compared to replication. An (n, k) systematic erasure code divides the original data into *stripes*, each containing k plaintext data blocks and $(n - k)$ coded parity blocks. Each stripe is stored across distinct nodes for maximal fault tolerance. This guarantees that the original stripe can be reconstructed as long as no more than $(n - k)$ of its blocks are lost. The conventional approach stripes data into uniformly sized data blocks and then generates parity blocks of the same size. This ensures the optimal storage overhead of $\frac{(n-k)}{k}$. If a stripe contains variable-sized data blocks, erasure coding requires padding all other data blocks in that stripe to match the size of its largest data block and generates parity blocks of the same size. However, this can lead to increased storage overhead. Figure 2 illustrates the relationship between data block size and storage overhead. A $(9, 6)$ erasure code encodes a 12MB object into two stripes, each containing 6MB of data. Stripe 1 divides the data into fixed-sized blocks of 1MB, achieving the lowest storage overhead of 0.5× (i.e., $\frac{3 \cdot 1MB}{6MB}$). In stripe 2, data blocks have variable sizes and must be padded implicitly to match the largest data block size, which is 3MB, and parity blocks are generated accordingly. This results in a higher storage overhead of 1.5× (i.e., $\frac{3 \cdot 3MB}{6MB}$). Therefore, existing storage systems [76, 91, 94] typically partition data into fixed-sized blocks before applying erasure coding to minimize storage overhead.

Systematic Reed-Solomon (RS) is the most prevalent erasure code used in modern storage systems, with two most

Table 1. An example table of Employees.

name	salary
Alice	70000
Bob	80000
Charlie	70000
David	60000
Emily	60000
Frank	70000

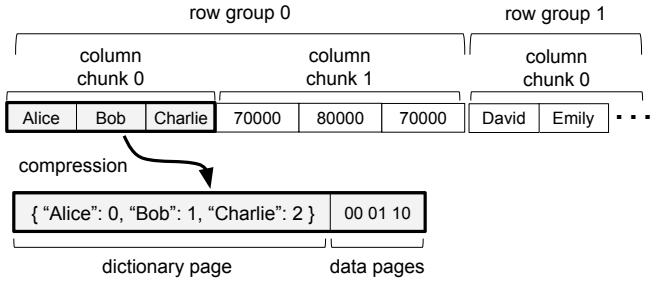


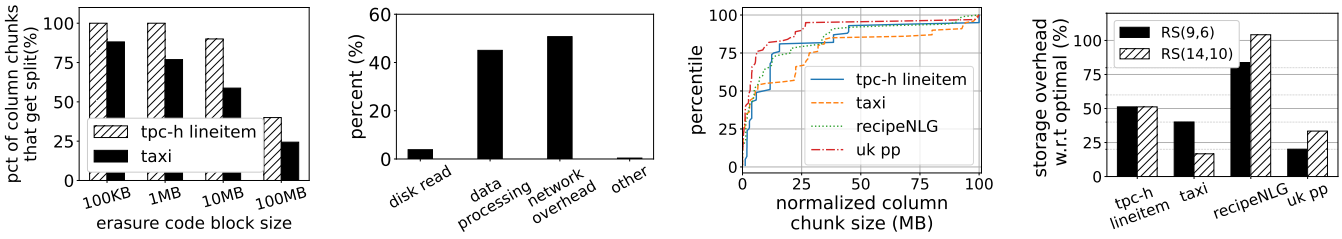
Figure 3. The PAX data layout. Column chunks are heavily compressed by default before being written to disk.

common configurations: (9,6) and (14, 10) [9, 92]. For the rest of this paper, we use $RS(9, 6)$ as the default erasure code.

Analytics file formats. Analytics systems predominantly use column-oriented storage formats, such as Apache Parquet [4] and ORC [39], due to their efficient data storage and retrieval. These columnar formats use a hybrid data layout called Partition Attributes Across (PAX) [38] that horizontally partitions a table into a set of row groups and then vertically partitions a row group column-wise into column chunks laid out contiguously on disk. Figure 3 depicts the PAX data layout of Table 1, using a row group size of 3 rows.

Compared to unstructured and semi-structured data formats such as CSV and JSON [28], PAX achieves better compression by grouping values of the same data type into column chunks. Parquet, by default, employs a variety of encoding and compression techniques (e.g., dictionary encoding, bit packing, and run-length encoding) on column chunks before writing them to disk. Each column chunk is a self-contained data unit for encoding and compression. As shown in Figure 3, the employee names from the first column chunk are first mapped into a sequence of unique integer codes: "0", "1" ,"2". Since there are only three distinct values, each integer code can be represented using only two bits. The names are further bit packed into a bit stream: "00", "01", "10". Finally, this column chunk is stored as a dictionary page followed by a number of encoded data pages on disk.

Moreover, PAX is efficient for columnar scans, filters, and projections as it retrieves only the necessary column chunks from disk. Consequently, a column chunk is the smallest



(a) (b) (c) (d)

Figure 4. (a) Percentage of column chunks that get split in RS(9,6) under various erasure code block sizes for TPC-H lineitem and taxi Parquet files. (b) Latency breakdown of a SQL query on TPC-H lineitem Parquet file under existing erasure-coded systems.(c) CDF of normalized column chunk sizes in real-world Parquet files (b) Storage overhead of the padding approach [36].

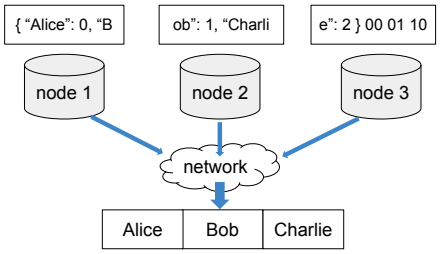


Figure 5. Processing a column chunk incurs non-trivial data-reassembly cost over the network.

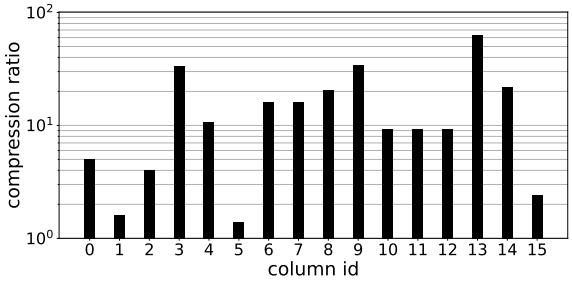


Figure 6. Average compression ratio of column chunks from each column in TPC-H lineitem Parquet file.

computable unit of PAX-based formats. For brevity, the remainder of the paper uses column chunks to denote the smallest computable units. In this work, we focus on the popular Parquet format as a case study, but our techniques are generalizable to other structured analytics file formats.

3 Motivation

In this section, we discuss why computation pushdown is fundamentally limited in existing erasure-coded systems and highlight the challenges involved in improving its efficiency.

3.1 Current Practices and Their Limitations

Existing storage system architectures typically implement the analytics file formats and erasure coding in separate layers of the software stack. Consequently, erasure codes operate without any awareness of the internal semantics of the highly structured objects, treating them merely as a blob of bytes. They divide objects into fixed-sized blocks, whose sizes are often pre-configured by the storage system, ranging from a few MBs to 100MBs to evenly distribute I/O [18, 29, 33, 74, 76, 81, 91, 94]. As a result, column chunks of an object may be split across many nodes. Figure 4a depicts the percentage of column chunks in two representative Parquet files, namely the TPC-H lineitem table [31] and the NYC taxi rides dataset [13], that get split in existing storage systems. We experiment over a range of erasure code block sizes as reported by real production systems. Notably, even with a large block size of 100MB, the percentage of column chunk splits remains significant, reaching up to 40% and 24% for the two datasets, respectively. As a result, the storage system must first reassemble a column chunk across storage nodes before processing.

Running example. We use the following query as a motivating example to illustrate this problem. Assume the company wants to know Bob's current salary from Table 1:

```
SELECT salary FROM Employees WHERE name == 'Bob'
```

As shown in Figure 5, column chunk 0 of row group 0 from Table 1 is split across three nodes. Although node 3 has the encoded data, it cannot directly run the WHERE clause to search for the name, Bob, since it cannot decode the encoded data without the dictionary. Unfortunately, the dictionary page is split across two other nodes. Therefore, the system must reassemble this column chunk across all three storage nodes before it can run the filter operation to find matching rows whose name is Bob. Similar steps are repeated for the salary column. This can incur non-trivial network latency.

We measure the cost of data reassembly in current storage systems by executing the microbenchmark query outlined in §6 on the TPC-H lineitem Parquet file. The SQL query retrieves about 1% of column values. Figure 4b shows the latency breakdown of this query. We find that the interested column chunks span across 5 storage nodes on average. Despite the large chunk sizes, disk reads contribute only a small fraction to the total query time. But 50% of the time is spent

on network operations for transferring and reassembling fragmented parts of the column chunks.

3.2 Challenges

The column chunk sizes are influenced by various factors such as the data type and data cardinality (i.e., the number of column values). In addition, most columnar data formats integrate compression and encoding to significantly reduce storage costs when writing column chunks to disk. Figure 6 shows the average compression ratio of column chunks from each column in the TPC-H lineitem Parquet file, with a median of 9.3 and a maximum of 63.5, respectively. The exact size of a column chunk also depends on the compressbility of its contents. Thus, column chunks within a PAX object can have variable sizes. Figure 4c shows the normalized size distribution of column chunks in four real-world Parquet files with compression enabled. Different datasets demonstrate distinct size distributions. For instance, in the TPC-H lineitem Parquet file, the majority of column chunks are either extremely small, representing highly repetitive integer values, or extremely large, representing diverse arbitrary string types. In contrast, the NYC taxi Parquet file exhibits a more uniform chunk size distribution since the dataset records user trip information that is more diverse in nature including pickup timestamp, trip duration, and distance.

To prevent splitting variable-sized column chunks across fixed-sized erasure code blocks, prior work [36] proposes a padding solution that inserts *additional* padding into the original objects to enforce the alignment of column chunks with the underlying storage blocks. If placing a column chunk in the current block would lead to splitting, the padding solution fills the remaining space in the current block with extra padding and relocates that column chunk to the next erasure code block. However, the extra padding results in increased storage overhead. As depicted in Figure 4d, we measure the storage overhead of the padding approach w.r.t. the optimal strategy when storing real-world Parquet files. Surprisingly, the additional storage overhead can be very high, in some cases exceeding 100%.

To summarize, the mismatch between variable column chunk sizes and fixed erasure code block sizes fundamentally limits the efficiency of computation pushdown in existing systems. This sub-optimality arises because file formats and erasure coding operate in separate layers of the storage stack, functioning within isolated silos. This motivates our design of Fusion, a new analytics object store that maximizes the benefits of computation pushdown, by co-designing its erasure coding to take into account the specific file format characteristics of the underlying data. Yet, incorporating file-format awareness into erasure coding is challenging, as seen in the padding solution, which incurs significant storage overhead. Fusion departs from the conventional fixed block-size-per-stripe approach used in most industry systems today. Instead, it utilizes a variable block-size-per-stripe approach that dynamically aligns erasure code blocks to column chunk boundaries within an object while minimizing storage overhead.

Table 2. Terminology.

Term	Description
column chunk	The smallest computable data unit in Parquet.
data/parity block	The smallest storage unit for data placement.
stripe	A set of data and parity blocks. The basic unit for data recovery and redundancy.
bin	Equivalent to a data block.
binset	Equivalent to all data blocks in a stripe.
FAC	Fusion's file-format-aware coding technique.

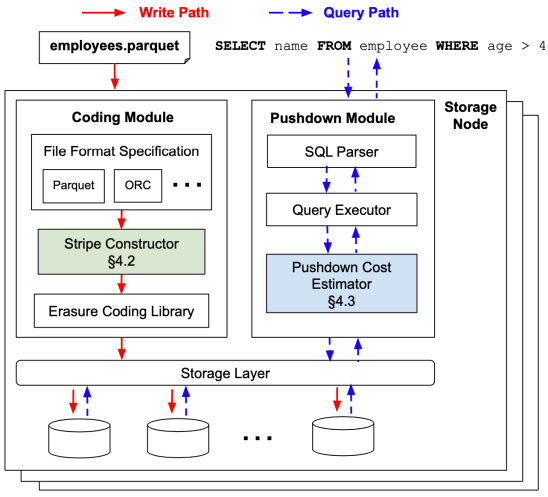


Figure 7. Fusion system components.

4 Fusion Design

This section discusses Fusion's system components. Table 2 summarizes the key terminology.

4.1 Design Overview

We outline the main design principles that underpin Fusion's architecture and provide an overview of its design.

Use variable block sizes to prevent fragmentation of column chunks. While it is possible to resize column chunks to fit some fixed block sizes (e.g., by adjusting the row group size in Parquet), this alternative approach has several drawbacks. First, it incurs high CPU overhead. Resizing a column chunk (e.g., rewriting a large chunk into smaller ones) requires decoding and re-encoding the original data, which is computationally expensive [82, 110]. Second, it is inflexible. If storage block sizes change in the future due to system upgrades or optimizations, all column chunks must be regenerated to align with the new sizes. Third, altering the data format (e.g., with a different row group size) may create inconsistencies and compatibility issues, as clients expect to retrieve objects in the same format in which they were

originally written to the storage service. To avoid these complications, Fusion stores column chunks in their original sizes without modification, and dynamically adjusts storage block sizes during erasure coding to prevent splitting any variable-sized column chunks.

Push down granular computations at the column chunk level to reduce network latency. A SQL query typically scans only a subset of table data (e.g., specific columns) and performs granular operations like filters and projections, which can run independently and in parallel on individual column chunks. Since Fusion stores each column chunk intact on a single storage node, its data layout is inherently optimized for computation pushdown at the column chunk level. Fusion utilizes a fine-grained pushdown mechanism that executes operations directly on the relevant column chunks and aggregates the computation results over the network. In contrast to existing object stores that read and reconstruct entire objects over the network, Fusion significantly reduces network latency by minimizing I/O and network usage.

Figure 7 depicts the high-level architecture of Fusion. It consists of a coding module (FAC) and a query pushdown module. When an object is uploaded to Fusion, FAC identifies the semantic boundaries of individual column chunks within the object based on its specified data format. It dynamically adjusts data block sizes per stripe to accommodate these variable-sized column chunks. It then constructs erasure code stripes and distribute them across storage nodes. FAC ensures that no column chunk is partitioned across data blocks, while simultaneously minimizing any additional storage overhead. When a SQL query arrives in Fusion, the query pushdown module transforms the original query into finer-grained operations (e.g., filters and projections). It estimates the pushdown cost of individual operations and selectively pushes down those operations to nodes where the relevant column chunks reside, provided that doing so leads to reduced latency. Finally, it orchestrates the execution of individual operations and constructs the final result back to clients. In Fusion, storage nodes are identical to each other and run both modules. Fusion does not have a dedicated coordinator. Each node acts as a coordinator for handling requests. Next, we dive deeper into the two modules.

4.2 File-Format-Aware Coding (FAC)

Drawing insights from §3, we introduce the novel concept behind the coding module called *file-format-aware coding* (FAC). FAC leverages knowledge of an object's internal structure during erasure coding, making sure that column chunks are never split across data blocks. However, since column chunks exhibit variable sizes (Figure 4c), a naive way to construct erasure code stripes may lead to uneven data block sizes, resulting in higher storage overhead. Figure 8 illustrates this problem. The stripe has a total data size of 6MB. The larger column chunks are placed in the first data block,

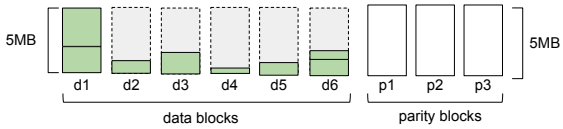


Figure 8. A stripe of variable-sized data blocks under RS(9,6). Green boxes are column chunks. Dashed grey boxes are padding.

d1, with a total size of 5MB. In this case, an erasure code must pad all data blocks to the size of the largest data block and generate parity blocks of the same size. Its storage overhead is 2.5×, whereas the optimal storage overhead is 0.5×, achieved by dividing the stripe into 6 1MB data blocks and generating 3 1MB parity blocks. Hence, the primary objective of FAC is to minimize this additional storage overhead in the presence of variable-sized data blocks. Next, we present a formal definition of this problem.

Problem Formulation. Based on Figure 8, we make a key observation: the size of parity blocks in a stripe depends *solely* on the largest data block size within the same stripe. Therefore, minimizing storage overhead is equivalent to minimizing the sum of the largest data block size from all stripes. We model this challenge as a variant of the bin packing problem. Specifically, we define each data block as a bin with capacity C, and all data blocks belonging to the same stripe as a *bin set*. Under a (n, k) erasure code, each bin set contains k bins. Given a list of N column chunks with sizes $\{s_1, \ldots, s_N\}$ and a total number of m bin sets, we want to assign column chunks into these bin sets such that the total sum of the largest bin size from all bin sets is minimized. Formally, we express the optimization objective as follows, where x_{ijl} are binary variables $\in \{0, 1\}$ with 1 indicating that the i-th column chunk is assigned to the j-th bin of the l-th bin set, and 0 otherwise:

$$\begin{aligned} \min_{x_{ijl}} \quad & \sum_{l=1}^{m} \max\{\sum_{i=1}^{N} s_i \cdot x_{ijl},\ \forall j \in \{1, \ldots, k\}\} \\ \text{subject to} \quad & \sum_{i=1}^{N} s_i \cdot x_{ijl} \le C,\ \forall j \in \{1, \ldots, k\}\ \forall l \in \{1, \ldots, m\} \\ & \sum_{l=1}^{m} \sum_{j=1}^{k} x_{ijl} = 1,\ \forall i \in \{1, \ldots, N\} \end{aligned} \tag{1}$$

The first constraint ensures that the total size of column chunks assigned to each bin does not exceed its capacity. The second constraint requires that each column chunk must be assigned to one and only one bin within some bin set. We set the bin capacity C as the maximum column chunk size such that each bin has sufficient space to accommodate any individual column chunk. Since each bin hosts at least one column chunk, we set m to be N/k, the maximum number of required bin sets.

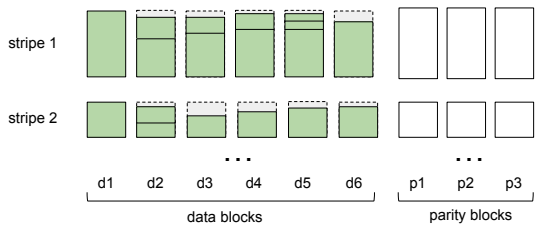


Figure 9. A visualization of the stripe construction algorithm. Green boxes are column chunks. Dashed grey boxes are padding.

Note that classic bin packing problems typically focus on minimizing the total number of bins used. After an extensive review of various bin packing variants, we found no exact match for our problem. The closest variant is minimizing the maximum bin size across all bins, also known as the multiprocessor scheduling problem [70]. However, this formulation differs from our objective, which aims to minimize the *sum* of the maximum bin sizes across all bin sets. To the best of our knowledge, our formulation introduces a new variant of the bin packing problem. §7 presents more related work on bin packing.

The Oracle. The above problem formulation belongs to the integer linear programming (ILP) class, which is known to be NP-complete. We implemented a solution using the Gurobi Optimizer [8], a popular commercial ILP solver. To assess its performance, we generated synthetic datasets by randomly sampling column chunk sizes between 1MB to 100MB. Figure 10a depicts its solve time as the number of column chunks increases. The ILP solver becomes impractically slow as problem size increases. Notably, with just 35 column chunks, Gurobi requires over 3 hours to compute an optimal solution. For context, real-world analytics files commonly comprise hundreds to thousands of column chunks, as illustrated in Table 3. Given that FAC operates on the critical path of `Put` operations, such excessive latency is unacceptable if this solver were to be integrated.

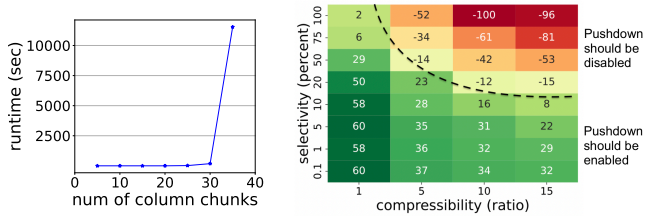
FAC Stripe Construction Algorithm. FAC uses an efficient, lightweight algorithm based on the insight that a stripe's storage overhead is determined once its largest data block is finalized. The algorithm constructs one stripe at a time and chooses the largest unassigned column chunk as the largest data block. It adheres to two principles when constructing the rest of the stripe. First, it proactively reduces overhead for future stripes by selecting larger column chunks for the current stripe. Doing so prevents these sizeable column chunks from becoming the largest data blocks in future stripes, incurring higher overhead. Second, it reduces wasted padding space within the current stripe by filling gaps in each data block, excluding the largest one, with smaller column chunks. This aligns the sizes of the remaining data blocks close to the largest block size.

Algorithm 1 Stripe Construction Algorithm

```
1:  Procedure ConstructStripes(k, items = {s_1, s_2, ..., s_N})
2:  sid ← 0, stripes ← ∅
3:  Sort(items)    // Sort item sizes in descending order
4:  while items not empty do
5:      bins ← InitABinSet(k)    // Open an empty bin set
6:      max ← Pop(items)
7:      Append(bins[0], max)    // Add the largest item to the first bin
8:      for item in items do
9:          bid ← Return the bin id of the least occupied bin where item
            fits, or −1 otherwise
10:         // Exclude the first bin
11:         if bid > 0 then
12:             Append(bins[bid], item)
13:         end if
14:     end for
15:     stripes[sid] ← bins    // Seal the current bin set
16:     sid ← sid + 1
17: end while
18: Return stripes
```

Algorithm 1 presents the detailed algorithm. It starts with a queue of N unassigned column chunks from an object and sorts them in descending order based on size. In each iteration, it opens a bin set of k empty bins, pops the largest column chunk from the head of the queue, and places it into the first bin. At this point, it seals the first bin and sets the bin capacity C to be its size. This ensures that all other bins cannot grow larger than the first bin. Next, it iterates over the queue. For each unassigned column chunk, it checks whether this column chunk can fit into any of the bins, excluding the first one. Among all bins with adequate space to accommodate the column chunk, it allocates the column chunk to the least occupied bin, aiming to achieve a more balanced distribution of load within the bin set. After a full scan of the queue, it seals the current bin set and proceeds to the next iteration. The above steps are repeated until all column chunks are assigned. At the end, the algorithm outputs m bin sets, each containing k bins, with each bin containing one or more column chunks. Figure 9 provides a visualization of the algorithm. During erasure coding, data blocks, along with parity blocks, are distributed across n randomly chosen storage nodes for each stripe.

The time complexity of the algorithm is $O(m \cdot N)$. It is linearly proportional to the number of column chunks within an object. We find that FAC stripe construction algorithm runs extremely fast for real-world analytics file objects (i.e., 10s to 100s of microseconds). In contrast, the total put latency of analytics file typically ranges between 10s to 100s of seconds. As an example, uploading an 11GB TPC-H Parquet file to the baseline storage system takes 34 seconds. In comparison, FAC's algorithm completes in just 500 microseconds, constituting a negligible 0.0015% of the overall put latency. We provide more details on the runtime of our algorithm under realistic workloads in §6.3.



(a) (b)

Figure 10. (a) Runtime of FAC using an ILP solver. (b) Pushdown trade-off in four TPC-H lineitem columns (c5, c0, c4 and c7). Each cell shows p50 latency improvement (%) of Fusion compared to a baseline system that splits column chunks.

The theoretical worst-case storage overhead of the algorithm under a (n, k) erasure code is $(n-k)$, which is the same as the replication approach that tolerates the same number of failures. This extreme scenario occurs in a stripe when one data block is very large, and the rest of the data blocks are negligibly small. Since the parity block size of a stripe always equals the size of its largest data block, the storage overhead of this stripe becomes comparable to that of replication. We introduce a system-level hyperparameter in Fusion, allowing users to specify the maximum additional storage overhead they can tolerate compared to the optimal. If the algorithm cannot construct stripes within the specified storage budget, Fusion defaults to erasure coding the object into fixed-sized blocks, similar to existing storage systems. §6.3 conducts a comprehensive evaluation of storage overhead across diverse synthetic and real-world datasets. Empirically, we find that FAC incurs an additional storage overhead of no more than 1.24% compared to the optimal scenario.

4.3 Fine-grained Adaptive Query Pushdown

The query pushdown module manages the execution of SQL queries on FAC-encoded objects. It decomposes a query into sub-queries for individual column chunks and uses a cost model to determine whether to enable or disable pushdown at the column chunk level, ensuring maximum pushdown efficiency. The pushed-down operations will run in-situ on the storage nodes where the relevant column chunks reside. Fusion employs this fine-grained adaptive pushdown mechanism because we find query pushdown does not always reduce the network traffic and improve query latency. In fact, pushdown efficiency depends on query selectivity (i.e., the amount of data a query returns) and the compression ratios of column chunks. To better understand this, we first describe how a SQL query is executed in Fusion with pushdown always enabled.

When a query arrives at a coordinator node in Fusion, it breaks down the query into fine-grained operations and executes them in two stages: a filter stage followed by a projection stage. In the filter stage, filter operations are first forwarded to storage nodes hosting the column chunks to be filtered. A storage node will read the column chunk from disk, decompress and decode it, and then run the filter(s) on the decoded values. It returns a bitmap of the filtered results to the coordinator. The coordinator waits for all filter bitmaps and consolidates them into a final bitmap. In the projection stage, the coordinator forwards the final bitmap to all storage nodes hosting the column chunks to be projected. Each storage node reads the column chunk from disk, decompresses and decodes it, and then selects the values whose corresponding bits are set in the bitmap. Each sends back all selected values in the uncompressed form to the coordinator. Finally, the coordinator assembles all projection replies and sends the final query result back to the client.

Our measurement on real-world datasets in Figure 6 shows that some columns are highly compressed, due to a combination of encoding and compression techniques used in PAX-based file formats. When query selectivity is high (i.e., returning many rows), the size of uncompressed values may greatly exceed the compressed column chunk size during the projection stage. This degrades query performance of Fusion since it sends more data over the network compared to existing systems that reassemble compressed column chunks on the coordinator. Figure 10b shows the parameter space where pushdown is effective. We define the compressibility of a column chunk as the ratio of its uncompressed size to its compressed size. For queries with low selectivity (common in cloud analytics), pushdown improves query latency significantly. However, for queries with high selectivity or column chunks with high compressibility, pushdown degrades the latency.

Therefore, Fusion adopts a fine-grained adaptive approach to query pushdown. Rather than pushing down the entire query to storage nodes, Fusion estimates which column chunks will benefit the most from query pushdown and enables pushdown only for those column chunks. Column chunks with unfavorable selectivity and compressibility values may incur high network costs. The coordinator will fetch them in compressed form and process them locally. The *Pushdown Cost Estimator* sub-module performs the cost estimation for column chunks. It decides to push down projections of a column chunk only when $selectivity \times compressibility < 1$, which we refer to as the Cost Equation.

The compressibility of a column chunk is estimated from the metadata of PAX-based objects (e.g., Parquet file footer). At the end of the the filter stage, the coordinator, knowing the exact query selectivity when constructing the final filter bitmap, can assess whether the benefit of pushing down projections on a column chunk outweighs the network overhead of transferring the uncompressed projection results. Specifically, it uses query selectivity and the compressibility of column chunks to estimate the size of uncompressed projection results and pushes down projection operations only when the size of uncompressed results is smaller than

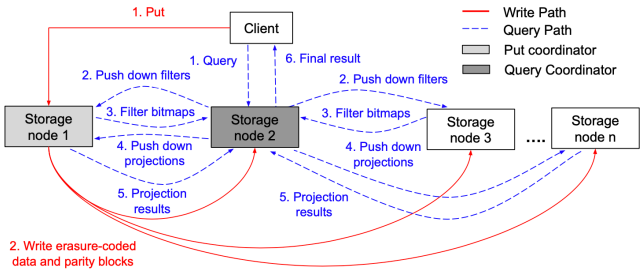


Figure 11. Fusion execution flow.

the encoded column chunk size. Otherwise, Fusion disables projection pushdown for this column chunk and reassembles it at the coordinator instead.

5 Implementation

Fusion is written in the Go language and uses Apache Arrow's Parquet library [2]. It supports three APIs, `Put`, `Get` and `Query`. `Put` writes an object to Fusion. `Get` retrieves the contents of the object with a given offset and size. `Query` runs a SQL query on the stored objects. We focus on `Put` and `Query` in this section as they are more interesting. Figure 11 shows the execution flow of `Put` and `Query` in Fusion. To achieve better load balancing, Fusion's implementation does not use a dedicated coordinator node for handling client requests. Instead, client requests are routed to a storage node based on the hash value of the object's name. Each node in Fusion acts as a coordinator for both `Put` and `Query`.

Storing Objects. When a `Put` request arrives at a coordinator node, the FAC module on the coordinator first parses the object's metadata and identifies the byte offsets and sizes of all column chunks within the object. The stripe constructor then uses this information to find an intelligent stripe layout, while keeping the storage overhead under a system-configurable threshold. If no layout satisfies the threshold, it outputs the default layout that may split column chunks. Finally, FAC constructs erasure code stripes based on the selected layout, using some conventional erasure coding algorithm (e.g., Reed-Solomon), and persists the data and parity blocks across storage nodes. Note that even though FAC is designed to keep individual column chunks intact, the object is still distributed across the storage nodes, preserving the benefits of IO and compute parallelism. Updates in Fusion are treated as fresh inserts.

Querying Objects. Fusion supports basic SQL query expressions on objects. When a `Query` request arrives at a coordinator, the SQL executor on the coordinator executes this query in two stages: the filter stage and projection stage. In the filter stage, the coordinator uses built-in object metadata to perform coarse-grained filtering and skips column chunks that do not satisfy the filter condition. For example, the Parquet file footer contains the min and max values for each column chunk within a row group, which speeds up filtering at the row group level. The coordinator then looks up Fusion's chunk location map to identify the storage nodes hosting the relevant column chunks and pushes down filter operations to those nodes. Each storage node constructs a bitmap for each column chunk, indicating the rows that satisfy the filter condition. It uses Snappy to compress bitmaps before sending them back to the coordinator. At the end of the filter stage, the coordinator aggregates all filter bitmaps and knows the exact query selectivity. The pushdown cost estimator uses the Cost Equation described in §4.3 to evaluate the potential benefits of pushing down projection operations. Projections on column chunks that meet the criteria are pushed to the relevant storage nodes. The remaining column chunks are fetched and processed locally at the coordinator. At the end, the coordinator consolidates all projection results and returns the final query result to the client.

Metadata Management Fusion keeps a chunk location map per object, tracking the storage node that hosts each column chunk. Every map entry is 8 bytes in size with 4 bytes for the column chunk offset within the object and 4 bytes for the storage node ID. The size of the location map is much smaller (a few KBs) compared to the overall object size, typically in GBs, as shown Table 3. Fusion must ensure the same durability for the location map as for the object itself because the system would be unable to retrieve the object if losing its map. In the current implementation, the coordinator replicates the location map to $k+1$ storage nodes to tolerate the same number of failures as RS(n, k). In the future, Fusion will store location maps in a distributed storage service like ZooKeeper [23] or etcd [27].

Recovery and Fault Tolerance. Fusion does not modify the erasure code algorithms and follows the conventional recovery procedures for data reconstruction when one or more data blocks are lost. Therefore, Fusion provides the same level of fault tolerance as existing erasure codes.

SQL Support. Fusion supports the `SELECT` SQL command and standard ANSI clauses i.e., `SELECT`, `FROM` and `WHERE` (similar to S3 Select [25]). It currently lacks support for aggregate pushdown such as `SUM` and `AVG`, which we aim to implement in the future to further enhance Fusion's performance.

Note that Fusion is designed as an analytics-optimized object store for ad-hoc, interactive queries. It is not a data warehouse solution like Snowflake [55] or Amazon Redshift [72], which are specialized for operational workloads. Therefore, complex queries such as joins are excluded from Fusion, as they better run in a data warehouse. Meanwhile, data warehouses can still leverage Fusion by pushing down parts of the query plans to accelerate performance. We envision Fusion as a object store that efficiently handles simple ad-hoc analytics queries and as pushdown engine that complements data warehousing systems.

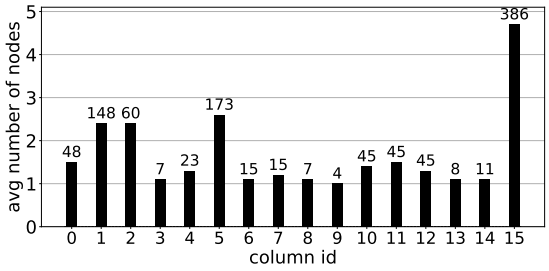


Figure 12. Average number of nodes a column chunk from TPC-H lineitem Parquet is stored on in baseline w/ column chunk split. Average column chunk size (MBs) shown on top of each bar.

6 Evaluation

We evaluate Fusion by answering the following questions:

- How does file-format-aware coding impact latency? (§6.1)
- How does Fusion perform on real-world SQL queries? (§6.2)
- What are the overheads of FAC? (§6.3)

Table 3. Parquet dataset file description

dataset	num columns	num column chunks	size (GB)
tpc-h lineitem	16	160	10
taxi	20	320	8.4
recipeNLG	7	84	0.98
uk pp	16	240	1.5

Configuration. All experiments are run on a cluster of r6525 cloudlab [6] machines. Each machine has 64 cores, 256 GB RAM, 1.6 TB NVMe SSD, 100GbE network and runs Ubuntu 22.04.3 LTS. We use a cluster of 10 machines, with one machine as a dedicated client node and the rest as storage nodes. All systems are evaluated using 10 clients and a total of 10 K queries. Erasure coding parameters are RS(9,6) and block size is set to 100 MB. We set the storage overhead threshold to 2% in Fusion which ensures that its additional storage overhead w.r.t. optimal never exceeds 2%. All disk I/O is done using direct I/O to avoid the effects of OS page caching on query latency. We use wondershaper [17] to limit the incoming and outgoing bandwidth on each machine to 25Gbps for all experiments, except in Figure 14c.

Table 4. Real-world SQL query description

query	dataset	num filters	num projections	select-ivity
Q1 (projection heavy)	tpc-h	1	6	1.4%
Q2 (filter heavy)	tpc-h	3	2	5.4%
Q3 (high selectivity)	taxi	1	1	37.5%
Q4 (low selectivity)	taxi	1	2	6.3%

Datasets. We use the TPC-H [31] and the NYC yellow taxi [13] datasets and convert them to Parquet format. TPC-H lineitem table Parquet file is 10GB in size. It consists of 16 columns and 10 row groups. Each row group is 1GB and contains 30M rows. NYC yellow taxi Parquet file uses dataset from years 2015-2017 and is 8.4GB. It has 20 columns and 16 row groups, each of size 525MB and 25M rows. See Table 3 for details. All parquet files are generated using pyarrow [14] v13.0.0 and Parquet format 2.4, and have dictionary encoding and Snappy compression enabled. We create two datasets of 100GB and 84GB, respectively, by duplicating each Parquet file ten times. While cloud datasets can reach terabytes in size, individual files are typically tens of gigabytes in size. The Parquet file sizes we use are consistent with the real-world.

Workloads. We run two sets of workloads: microbenchmark queries and real-world SQL queries. The microbenchmark executes a SQL query on the TPC-H lineitem table to retrieve a single column with a filter condition. Query selectivity by default is 1%, as observed in production systems [59, 85]. The only exception is the selectivity sweep evaluation in §6.1, where we vary query selectivity by changing the `value` field:

```
SELECT column FROM lineitem WHERE column < value
```

We also use four real-world SQL queries based on the TPC-H and taxi datasets (Table 4). Q1 and Q2 are the pricing summary report query and forecasting revenue change query from TPC-H [15]. Q3 and Q4 are two analytics queries from Timescale [12] on the taxi dataset, as shown below.

(1) How many rides took place every day in 2015?

```
SELECT count(*) FROM taxi WHERE date < 2015-12-31
```

(2) What is the average fare amount in January 2015?

```
SELECT date,AVG(fare) FROM taxi WHERE date<2015-02-01
```

Baseline. We implement a baseline system representative of state-of-the-art systems such as MinIO [35] and Ceph [103], which erasure codes an object into fixed-sized blocks and may split its column chunks. The baseline assembles the Parquet object on a coordinator node before executing the query on it. It also uses the optimization that leverages Parquet file metadata (i.e., the file footer) to retrieve only those column chunks that match the SQL query filter condition.

Performance metrics. We focus on *median* and *tail* latency in the performance analysis. Fusion targets at ad-hoc, interactive analytics queries where latency is the primary metric of interest. Leading industrial analytics systems such as Athena and BigQuery often report query completion time in their performance blogs [20–22]. Similarly, data analytics frameworks like Spark [109] and MapReduce [56] prioritize job completion time over throughput in their evaluations. In the context of data analytics, latency is a more meaningful metric due to the high variance in query runtime.

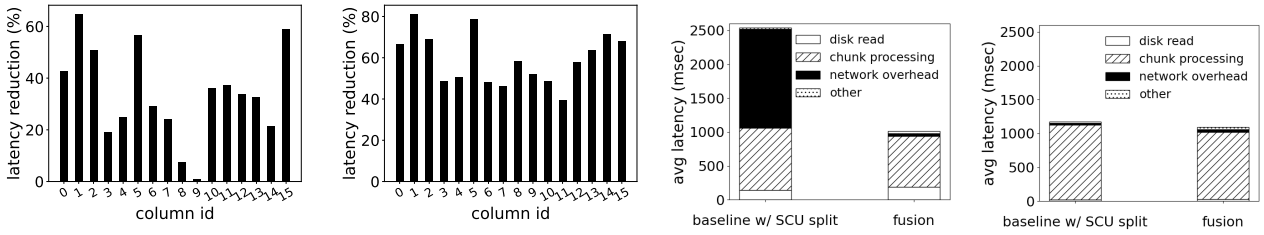


Figure 13. (a) p50 and (b) p99 latency reduction for TPC-H lineitem columns. Latency breakdown of (c) column 5 and (d) column 9.

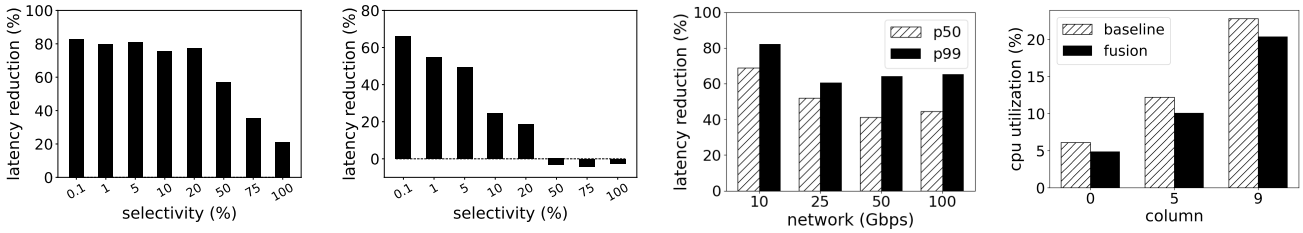


Figure 14. Impact of query selectivity on tail latency for (a) column 5 and (b) column 9. (c) Network bandwidth sweep for column 5. (d) Averaged CPU utilization per node.

6.1 Micro-benchmarks: Impact of FAC on Latency

Column sweep. We first show results of microbenchmark queries on individual columns of the TPC-H lineitem Parquet file. Figure 13a and 13b shows median and tail latency improvement per column achieved Fusion compared to the baseline. Queries on columns 0, 1, 2, 5, and 15 show up to 65% and 81% for median and tail latency reduction. As shown in Figure 12, these column chunks are bigger in size and are more likely to get split in the baseline. As a result, the baseline must perform parallel reads from multiple storage nodes and transfer a substantial amount of data over the network. We closely examine one column as a case study to better understand why Fusion outperforms the baseline in this scenario. Figure 13c shows the query latency breakdown of column 5. Disk read involves the time spent reading the raw data from disk, while chunk processing encompasses the total time required for decoding data from Parquet format and evaluating SQL operations. Network overhead constitutes the combined latency from network transfer and the additional overhead introduced by remote procedure calls (RPCs). The chunks of column 5 are large in size (i.e., 165MB) and span across 2.6 storage nodes on average. Both system spend approximately the same amount of time on disk read and chunk processing. However, the baseline spends about 57% of its total time to reassemble column chunks over the network compared to Fusion, which processes the column chunks *in-situ* on the storage nodes and takes less than 4% of its total time to transfer the final filtered result (i.e., 1% of all column values). In this case, the baseline transfers 2827GB of data over the network compared to Fusion, which transfers just 44GB, i.e., a 64.2× reduction in network traffic.

On the other hand, the latency improvements for columns 3, 4, 9, 10, and 11 are modest. These column chunks are smaller in size and exhibit higher compression ratio (see Figure 6). Consequently, the baseline system can read these chunks over the network, often from a single storage node, without incurring substantial network traffic. Figure 13d shows the latency breakdown for column 9. Both systems spend no more than 3% of their total time on network operations. We find that Fusion consistently achieves better performance gains under low query selectivity. In contrast, the performance of the baseline system varies significantly and depends on a number of factors, including the column type and individual chunk sizes.

Query selectivity. Next, we vary query selectivity to understand its impact on latency. We conduct a selectivity sweep for column 5 and column 9, representing one of the better-performing columns and one of the worst-performing columns, respectively, in Fusion. Figure 14a and 14b shows the latency reduction of Fusion compared to the baseline. Recall the Cost Equation from §4.3. Fusion is expected to achieve the most performance gains when query selectivity and/or the compressibility of column chunks is low. The results support this intuition, as queries with lower selectivity exhibit the highest latency gains. Under high selectivity, such as 75% and 100%, Fusion identifies that the size of uncompressed projection result significantly exceeds that of the original compressed chunk to be projected. Hence, Fusion disables projection pushdown for the column chunk

and falls back to the baseline mode, transferring the compressed chunk to the coordinator for further processing. It's worth noting that even under these extreme cases, Fusion still benefits from the pushdown of filter operations.

Network. Figure 14c shows the median and tail latency reduction of Fusion compared to the baseline under various network configurations. Fusion achieves higher latency gains under a more constrained network (e.g., 10Gbps). Due to its high data transfer volume, the baseline experiences more performance degradation on a slower network.

CPU utilization. Figure 14d compares the CPU utilization of both systems for different columns under a fixed system load of 10 queries per sec. The CPU utilization is averaged across all storage nodes. Despite performing the same amount of computation work, Fusion utilizes less CPU since it transfers less data over the network. The efficient computation pushdown in Fusion enables it to save valuable CPU resources while delivering the same system throughput as the baseline.

6.2 Performance on Real-World SQL Queries

Figure 15a illustrates the median and tail latency reduction of Fusion on real-world SQL queries. For Q1 and Q2, Fusion achieves a median latency reduction of up to 48% and a tail latency reduction of up to 40%. Both queries involve filter and projection operations across multiple columns. Fusion's file-format-aware coding approach (FAC) enables the direct pushdown of filter and projection operations onto the storage nodes hosting the column chunks, effectively reducing network overheads. On taxi-based queries, Fusion reduces the median latency by up to 32% and tail latency by up to 48%. For Q3, despite the high selectivity (i.e., 37.5%), the `date` column has a low compression ratio of 1.6. Fusion still offers latency gains since the selectivity and compressibility product is 0.6, which is less than 1, making it suitable for pushdown according to the Cost Equation. Q4 involves two projection columns, namely `date` and `fare`. However, the `fare` column has a high compression ratio of 152. The product of selectivity and compressibility for this column far exceeds 1, leading Fusion to disable the projection pushdown. Despite this, Fusion still outperforms the baseline due to its effective pushdown of the `date` column. Figure 15b shows the total network traffic usage in Fusion compared to the baseline system. Fusion generates upto 8.9× lower network traffic due to its pushdown friendly data layout.

6.3 Overheads of FAC

Next we evaluate the storage and runtime overhead of FAC on both synthetic and real-world datasets. The optimal storage overhead under RS(9,6) is 0.5× (i.e., $\frac{9-6}{6}$). We report the storage overhead of FAC as a percentage, representing its additional overhead relative to the optimal. The synthetic datasets simulate a wide range of file configurations. allowing for a comprehensive evaluation of storage overhead.

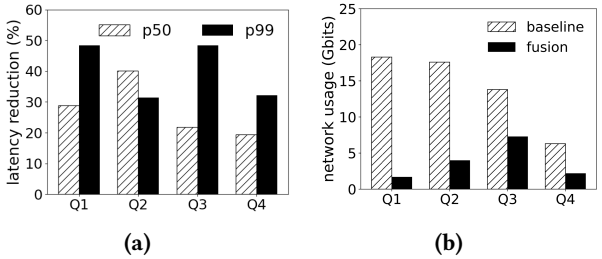
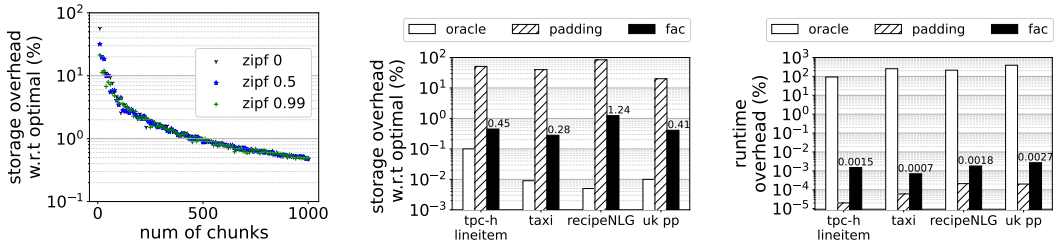


Figure 15. Performance on real-world SQL queries.

Specifically, we vary the number of column chunks and the skewness of the chunk size distribution. Each synthetic dataset consists of chunk sizes between 1MB to 100MB, chosen from a Zipfian distribution. Each data point in Figure 16a is the averaged storage overhead over 100 dataset runs. Surprisingly, we find that the chunk size distribution has little impact on FAC's storage overhead. When the chunk sizes are highly skewed (e.g., Zipfian 0.99), the FAC algorithm can effectively group the majority of chunks with similar sizes together. The remaining are packed into a small number of uneven data blocks, with negligible storage overhead. When chunk sizes are random (e.g., Zipfian 0), the algorithm benefits from high variance and can distribute chunks more uniformly across data blocks, increasing the likelihood of bin-packing into similar-sized blocks. We find that the storage efficiency of FAC is directly influenced by the number of chunks. A higher number of chunks results in a larger search space, offering more satisfactory solutions and increasing the likelihood of finding a good one. As illustrated in Figure 16a, FAC's storage overhead decreases to 3% with 100 chunks, drops to 0.8% with 500 chunks and approaches closer to 0 for higher values. For large objects with more column chunks, the storage overhead of FAC tends to converge towards the optimal. FAC's storage overhead under RS(14, 10) exhibits a similar pattern, which we omit here due to space limits.

Next, we measure the storage overhead of FAC on four real-world Parquet files, with the number of column chunks ranging from 84 to 320 (Table 3). We compare FAC against two alternative approaches: `Oracle`, the Gurobi-based ILP solution, and `Padding`, proposed by Adams et al. [36], which inserts extra padding into the original object to enforce data alignment with erasure code blocks. Figure 16b shows the additional storage overhead. Figure 16c shows the runtime overhead relative to the the total time of the `Put` operation when writing the file to the baseline storage system. While `Oracle` achieves minimal storage overhead, its runtime is prohibitively high, up to 3.91× the overall `Put` latency for uk pp. `Padding` runs faster but incurs a substantial storage overhead of up to 83.8% for recipeNLG. In contrast, FAC's stripe construction algorithm strikes a more favorable balance between storage overhead and runtime. It incurs a storage overhead of no more than 1.24% relative to the optimal and a runtime overhead of no more than 0.0027%, respectively.



(a) (b) (c)

Figure 16. (a) Storage overhead of FAC for different chunk size distributions. (b) and (c) Storage and runtime overhead (both in log scale) for real-world Parquet files. All plots assume RS(9,6).

Previous studies [47, 93, 99] suggest that large file objects in gigabytes dominate cloud storage. Our empirical findings lead us to believe that Fusion would result in little extra storage overhead when deployed in production.

7 Related Work

Computation Pushdown. Both systems and database research has actively explored computation pushdown techniques. Recent systems [43, 79, 107] enable the pushdown of arbitrary functions to storage nodes, reducing network round trips, particularly for multi-hop traversal queries like B-tree index lookups. Unlike Fusion, they are in-memory systems and do not consider erasure coding. New computational storage devices (CSDs) [30, 36, 58] allow application code to run directly inside the disk. Adams et al. [36] propose an application-level solution that pads objects to align their internal data units with erasure code blocks but may incur significant storage overhead. In contrast, Fusion operates within the storage layer and minimizes storage overhead. Cloud databases also employ predicate pushdown to speed up queries and reduce network traffic [7, 24, 26, 72, 105, 106, 108]. FlexPushdownDB [105] takes a hybrid approach that combines selective caching and query pushdown. It does not address pushdown efficiency in storage systems. To the best of our knowledge, Fusion is the first analytics object store that improves computation pushdown on structured analytics files stored in an erasure-coded form.

Analytics file formats. Analytics file formats can be categorized as unstructured (plaintext, binary files), semi-structured (CSV, JSON [28]), and structured ([3, 4, 39]). PAX-based [38] structured file formats, such as Parquet [4] and ORC [39], are widely used in data analytics. Albis [102] is another structured format that disables compression and encoding for better query performance. Fusion leverages the structural knowledge embedded in these formats to erasure code data into a computation-friendly layout.

Erasure coding. Systematic erasure codes are widely deployed in public clouds [1, 19, 35, 42, 62, 76, 80, 87, 100, 103] due to their advantage of storing data blocks in plaintext. Fusion currently supports systematic Reed-Solomon (RS) and can easily incorporate other systematic codes. Non-systematic codes are not supported since they store data blocks in encoded form, making direct computations on storage nodes infeasible. Prior work on erasure codes primarily focuses on improving repair performance through new classes of codes, such as regenerating codes [57] and locally repairable codes [69, 75, 76, 90, 95], which are orthogonal to the focus of Fusion.

Bin packing and its variants. The bin packing problem has been extensively studied for decades and has numerous variants, including heterogeneous bin types [65], variable bin sizes [44, 53, 64, 77, 96, 97], item fragmentation [61, 84, 86, 98], and different optimization objectives [40, 51, 54, 68] and constraints [37, 41, 63, 66, 78]. The problem space can be categorized into one-dimensional [50, 52, 67] and multidimensional [46, 48, 49, 60]. In the multidimensional case, item sizes and bin capacities span multiple dimensions and must satisfy constraints across all dimensions. The bin packing problem has many important applications in the cloud and data centers, such as resource-constrained scheduling [45, 66, 71, 88], and virtual machine placement [83, 89, 101]. These applications usually use online approximation algorithms since items arrive on-demand, and future items are not known in advance. Fusion tackles a new variant of the one-dimensional bin packing problem. It utilizes a greedy-based offline algorithm, as all items (i.e., column chunks) become available once an analytics object is uploaded.

8 Summary

Modern erasure-coded cloud object stores are not optimized for computation pushdown. By co-designing erasure coding and file placement topology, Fusion significantly reduces query latency for big data analytics with low storage overhead. We believe Fusion's co-design approach can benefit other system areas that use erasure codes for data storage.

Acknowledgments

We thank our shepherd Laurent Bindschaedler and the anonymous reviewers for their valuable feedback and suggestions. This research was supported by NSF awards CNS #2106530.

References

[1] [n. d.]. Amazon S3. https://aws.amazon.com/s3/.

[2] [n. d.]. Apache arrow parquet. https://arrow.apache.org/docs/python/parquet.html.

[3] [n. d.]. Apache Avro. https://avro.apache.org/.

[4] [n. d.]. Apache Parquet. https://parquet.apache.org/.

[5] [n. d.]. Azure Data Lake Storage Query Acceleration. https://learn.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-query-acceleration.

[6] [n. d.]. Cloudlab. https://www.cloudlab.us.

[7] [n. d.]. Filtering and retrieving data using Amazon S3 Select. https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html.

[8] [n. d.]. gurobipy 11.0.0. https://pypi.org/project/gurobipy/.

[9] [n. d.]. HDFS erasure coding. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[10] [n. d.]. Lance: modern columnar data format for ML. https://lancedb.github.io/lance/.

[11] [n. d.]. MinIO Select API Quickstart Guide. https://github.com/minio/minio/blob/master/docs/select/README.md.

[12] [n. d.]. NYC yellow taxi dataset queries. https://docs.timescale.com/tutorials/latest/nyc-taxi-cab/query-nyc/.

[13] [n. d.]. NYC yellow taxi trip record data. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[14] [n. d.]. pyarrow. https://pypi.org/project/pyarrow/.

[15] [n. d.]. TPC-H specification. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf.

[16] [n. d.]. Using Lua in the Ceph distributed storage system. https://www.lua.org/wshop17/Watkins.pdf.

[17] [n. d.]. Wondershaper. https://github.com/magnific0/wondershaper.

[18] 2012. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast12/rethinking-erasure-codes-cloud-file-systems-minimizing-io-recovery-and-degraded

[19] 2017. Backblaze Vaults: Zettabyte-Scale Cloud Storage Architecture. ttps://www.backblaze.com/blog/vault-cloud-storage-architecture/.

[20] 2021. Benchmarking Cloud Data-warehouse Bigquery to Scale Fast. https://cloud.google.com/blog/products/data-analytics/benchmarking-cloud-data-warehouse-bigquery-to-scale-fast.

[21] 2021. Run queries 3x faster with up to 70% cost savings on the latest Amazon Athena engine. https://aws.amazon.com/blogs/big-data/run-queries-3x-faster-with-up-to-70-cost-savings-on-the-latest-amazon-athena-engine/.

[22] 2022. Upgrade to Athena engine version 3 to increase query performance and access more analytics features. https://aws.amazon.com/blogs/big-data/upgrade-to-athena-engine-version-3-to-increase-query-performance-and-access-more-analytics-features/.

[23] 2023. Apache Zookeeper. https://zookeeper.apache.org.

[24] 2023. AQUA (Advanced Query Accelerator) for Amazon Redshift. https://pages.awscloud.com/AQUA_Preview.html/.

[25] 2023. AWS S3 select command. https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-select-sql-reference-select.html.

[26] 2023. Azure Data Lake Storage query acceleration. https://docs.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-query-acceleration/.

[27] 2023. etccd. https://etcd.io/.

[28] 2023. JavaScript Object Notation. https://www.json.org/json-en.html.

[29] 2023. Minio erasure coding parameters. https://blog.min.io/erasure-coding-cpu-utilization/.

[30] 2023. Samsung SmartSSD Computational Storage. https://news.samsung.com/global/samsung-electronics-develops-second-generation-smartssd-computational-storage-drive-with-upgraded-processing-functionality.

[31] 2023. TPC-H benchmark. https://www.tpc.org/tpch/.

[32] [n. d.]. Amazon Athena. https://aws.amazon.com/athena.

[33] [n. d.]. Ceph erasure coding parameters. https://docs.ceph.com/en/quincy/rados/operations/erasure-code-profile/.

[34] [n. d.]. Google Cloud BigQuery. https://cloud.google.com/bigquery.

[35] [n. d.]. MinIO: High Performance Object Storage for Modern Data Lakes. https://min.io/.

[36] Ian F. Adams, Neha Agrawal, and Michael P. Mesnier. 2021. Enabling Near-Data Processing in Distributed Object Storage Systems. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems* (Virtual, USA) *(HotStorage '21)*. Association for Computing Machinery, New York, NY, USA, 28–34. https://doi.org/10.1145/3465332.3470881

[37] Micah Adler, Phillip Gibbons, and Yossi Matias. 2002. Scheduling Space-Sharing for Internet Advertising. *Journal of Scheduling* 5 (03 2002). https://doi.org/10.1002/jos.74

[38] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. 2002. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal* 11 (nov 2002), 198–215. https://doi.org/10.1007/s00778-002-0074-9

[39] ORC Apache. 2018. Apache ORC: High-Performance Columnar Storage for Hadoop.

[40] S. F. Assmann, D. S. Johnson, D. J. Kleitman, and J. Y.T. Leung. 1984. On a dual version of the one-dimensional bin packing problem. *Journal of Algorithms* 5, 4 (Dec. 1984), 502–525. https://doi.org/10.1016/0196-6774(84)90004-X

[41] J Baewicz and Klaus H. Ecker. 1983. A linear time algorithm for restricted bin packing and scheduling problems. *Operations Research Letters* 2 (1983), 80–83. https://api.semanticscholar.org/CorpusID:122991098

[42] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony Rowstron. 2014. Pelican: A Building Block for Exascale Cold Data Storage. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 351–365. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/balakrishnan

[43] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. 2020. Adaptive Placement for In-memory Storage Functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 127–141. https://www.usenix.org/conference/atc20/presentation/bhardwaj

[44] Rainer E. Burkard and Guochuan Zhang. 1997. Bounded space online variable-sized bin packing. *Acta Cybernetica* 13, 1 (Jan. 1997), 63–76. https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3480

[45] Franck Butelle, Laurent Alfandari, Camille Coti, Lucian Finta, Lucas Létocart, Gérard Plateau, Frédéric Roupin, Antoine Rozenknop, and Roberto Calvo. 2016. Fast Machine Reassignment. *Annals of Operations Research* 242 (07 2016). https://doi.org/10.1007/s10479-015-2082-3

[46] Chandra Chekuri and Sanjeev Khanna. 2004. On Multi-dimensional Packing Problems. *SIAM J. Comput.* 33, 4 (2004), 837–851. https://doi.org/10.1137/S0097539799356265 arXiv:https://doi.org/10.1137/S0097539799356265

[47] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. 2017. Giza: Erasure Coding Objects across Global Data Centers. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 539–551. https://www.usenix.org/conference/atc17/technical-sessions/presentation/chen-yu-lin

[48] Henrik I. Christensen, A. Khan, Sebastian Pokutta, and Prasad Tetali. 2016. Multidimensional Bin Packing and Other Related Problems : A Survey. https://api.semanticscholar.org/CorpusID:9048170

[49] Henrik I. Christensen, Arindam Khan, Sebastian Pokutta, and Prasad Tetali. 2017. Approximation and online algorithms for multidimensional bin packing: A survey. *Computer Science Review* 24 (2017), 63–79. https://doi.org/10.1016/j.cosrev.2016.12.001

[50] Edward G. Coffman, János Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. 2013. *Bin packing approximation algorithms: Survey and classification.* Vol. 1-5. Springer, 455–531. https://doi.org/10.1007/978-1-4419-7997-1_35

[51] E. G. Coffman, J. Y.T. Leung, and D. W. Ting. 1978. Bin packing: Maximizing the number of pieces packed. *Acta Informatica* 9, 3 (Sept. 1978), 263–271. https://doi.org/10.1007/BF00288885

[52] E. G. Coffman, M. R. Garey, and D. S. Johnson. 1996. *Approximation algorithms for bin packing: a survey.* PWS Publishing Co., USA, 46–93.

[53] J. Csirik. 1989. An on-line algorithm for variable-sized bin packing. *Acta Inf.* 26, 8 (oct 1989), 697–709. https://doi.org/10.1007/BF00289157

[54] J. Csirik and V. Totik. 1988. Online algorithms for a dual version of bin packing. *Discrete Applied Mathematics* 21, 2 (1988), 163–167. https://doi.org/10.1016/0166-218X(88)90052-2

[55] 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. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[56] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[57] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. 2010. Network Coding for Distributed Storage Systems. *IEEE Transactions on Information Theory* 56, 9 (2010), 4539–4551. https://doi.org/10.1109/TIT.2010.2054295

[58] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1221–1230. https://doi.org/10.1145/2463676.2465295

[59] Pavan Edara and Mosha Pasumansky. 2021. Big Metadata: When Metadata is Big Data. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3083–3095. https://doi.org/10.14778/3476311.3476385

[60] Leah Epstein and Meital Levy. 2010. Dynamic multi-dimensional bin packing. *J. of Discrete Algorithms* 8, 4 (dec 2010), 356–372. https://doi.org/10.1016/j.jda.2010.07.002

[61] Leah Epstein and Rob van Stee. 2023. Approximation Schemes for Packing Splittable Items with Cardinality Constraints. In *Approximation and Online Algorithms: 5th International Workshop, WAOA 2007, Eilat, Israel, October 11-12, 2007. Revised Papers* (Eilat, Israel). Springer-Verlag, Berlin, Heidelberg, 232–245. https://doi.org/10.1007/978-3-540-77918-6_19

[62] Andrew Fikes. 2010. Storage Architecture and Challenges. https://cloud.google.com/files/storage_architecture_and_challenges.pdf.

[63] Luke Finlay and Prabhu Manyem. 2005. Online LIB problems: Heuristics for Bin Covering and lower bounds for Bin Packing. *http://dx.doi.org/10.1051/ro:2006001* 39 (07 2005). https://doi.org/10.1051/ro:2006001

[64] D. K. Friesen and M. A. Langston. 1986. Variable Sized Bin Packing. *SIAM J. Comput.* 15, 1 (1986), 222–230. https://doi.org/10.1137/0215016 arXiv:https://doi.org/10.1137/0215016

[65] Michael Gabay and Sofia Zaourar. 2015. Vector bin packing with heterogeneous bins: application to the machine reassignment problem. *Annals of Operations Research* 242 (2015), 161 – 194. https://api.semanticscholar.org/CorpusID:42722

[66] M. R. Garey, Ron Graham, David S. Johnson, and Andrew Chi-Chih Yao. 1976. Resource Constrained Scheduling as Generalized Bin Packing. *J. Comb. Theory A* 21 (1976), 257–298. https://api.semanticscholar.org/CorpusID:205025075

[67] M. R. Garey, R. L. Graham, and J. D. Ullman. 1972. Worst-case analysis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing* (Denver, Colorado, USA) *(STOC '72)*. Association for Computing Machinery, New York, NY, USA, 143–150. https://doi.org/10.1145/800152.804907

[68] Martin Josef Geiger. 2008. Bin Packing Under Multiple Objectives - a Heuristic Approximation Approach. arXiv:0809.0755 [cs.AI] https://arxiv.org/abs/0809.0755

[69] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. 2012. On the Locality of Codeword Symbols. *IEEE Transactions on Information Theory* 58, 11 (2012), 6925–6934. https://doi.org/10.1109/TIT.2012.2208937

[70] R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 17, 2 (mar 1969), 416–429. https://doi.org/10.1137/0117039

[71] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (aug 2014), 455–466. https://doi.org/10.1145/2740070.2626334

[72] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[73] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. 2021. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 233–248. https://www.usenix.org/conference/fast21/presentation/hu

[74] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. 2021. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 233–248. https://www.usenix.org/conference/fast21/presentation/hu

[75] Cheng Huang, Minghua Chen, and Jin Li. 2013. Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems. *ACM Trans. Storage* 9, 1, Article 3 (mar 2013), 28 pages. https://doi.org/10.1145/2435204.2435207

[76] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 15–26. https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang

[77] Jangha Kang and Sungsoo Park. 2003. Algorithms for the variable sized bin packing problem. *European Journal of Operational Research* 147, 2 (2003), 365–372. https://doi.org/10.1016/S0377-2217(02)00247-3 Fuzzy Sets in Scheduling and Planning.

[78] K. L. Krause, V. Y. Shen, and H. D. Schwetman. 1975. Analysis of Several Task-Scheduling Algorithms for a Model of Multiprogramming Computer Systems. *J. ACM* 22, 4 (oct 1975), 522–550. https://doi.org/10.1145/321906.321917

[79] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2018. Splinter: Bare-Metal Extensions for

Multi-Tenant Low-Latency Storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 627–643. https://www.usenix.org/conference/osdi18/presentation/kulkarni

[80] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14. https://doi.org/10.1109/MSST.2015.7208288

[81] Xiaolu Li, Runhui Li, Patrick P. C. Lee, and Yuchong Hu. 2019. OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 331–344. https://www.usenix.org/conference/fast19/presentation/li

[82] Yinan Li, Jianan Lu, and Badrish Chandramouli. 2023. Selection Pushdown in Column Stores using Bit Manipulation Instructions. *Proc. ACM Manag. Data* 1, 2, Article 178 (jun 2023), 26 pages. https://doi.org/10.1145/3589323

[83] Lei Lu, Hui Zhang, Evgenia Smirni, Guofei Jiang, and Kenji Yoshihira. 2013. Predictive VM Consolidation on Multiple Resources: Beyond Load Balancing. *IEEE International Workshop on Quality of Service, IWQoS*. https://doi.org/10.1109/IWQoS.2013.6550268

[84] C.A. Mandal, P.P. Chakrabarti, and S. Ghose. 1998. Complexity of fragmentable object bin packing and an application. *Computers & Mathematics with Applications* 35, 11 (1998), 91–97. https://doi.org/10.1016/S0898-1221(98)00087-X

[85] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[86] Nir Menakerman and Raphael Rom. 2001. Bin Packing with Item Fragmentation. 313–324.

[87] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. 2014. f4: Facebook's Warm BLOB Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 383–398. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muralidhar

[88] Aniket Murhekar, David Arbour, Tung Mai, and Anup Rao. 2023. Dynamic Vector Bin Packing for Online Resource Allocation in the Cloud. arXiv:2304.08648 [cs.DS] https://arxiv.org/abs/2304.08648

[89] Rina Panigrahy, Kunal Talwar, Lincoln K. Uyeda, and Udi Wieder. 2011. Heuristics for Vector Bin Packing. https://api.semanticscholar.org/CorpusID:17946270

[90] Dimitris S. Papailiopoulos and Alexandros G. Dimakis. 2014. Locally Repairable Codes. *IEEE Transactions on Information Theory* 60, 10 (2014), 5843–5855. https://doi.org/10.1109/TIT.2014.2325570

[91] K.V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (aug 2014), 331–342. https://doi.org/10.1145/2740070.2626325

[92] K.V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 331–342. https://doi.org/10.1145/2619239.2626325

[93] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 401–417. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/rashmi

[94] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2013. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/hotstorage13/workshop-program/presentation/rashmi

[95] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. XORing Elephants: Novel Erasure Codes for Big Data. *Proc. VLDB Endow.* 6, 5 (mar 2013), 325–336. https://doi.org/10.14778/2535573.2488339

[96] Steven S. Seiden. 2001. An Optimal Online Algorithm for Bounded Space Variable-Sized Bin Packing. *SIAM Journal on Discrete Mathematics* 14, 4 (2001), 458–470. https://doi.org/10.1137/S0895480100369948 arXiv:https://doi.org/10.1137/S0895480100369948

[97] Steven S. Seiden, Rob van Stee, and Leah Epstein. 2003. New Bounds for Variable-Sized Online Bin Packing. *SIAM J. Comput.* 32, 2 (2003), 455–469. https://doi.org/10.1137/S0097539702412908 arXiv:https://doi.org/10.1137/S0097539702412908

[98] Hadas Shachnai, Tami Tamir, and Omer Yehezkely. 2008. Approximation Schemes for Packing with Item Fragmentation. *Theor. Comp. Sys.* 43, 1 (mar 2008), 81–98. https://doi.org/10.1007/s00224-007-9082-x

[99] Yingdi Shan, Kang Chen, Tuoyu Gong, Lidong Zhou, Tai Zhou, and Yongwei Wu. 2021. Geometric Partitioning: Explore the Boundary of Optimal Erasure Code Repair. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 457–471. https://doi.org/10.1145/3477132.3483558

[100] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[101] Aneeba Khalil Soomro, Mohammad Arshad Shaikh, and Hameedullah Kazi. 2017. FFD Variants for Virtual Machine Placement in Cloud Computing Data Centers. *International Journal of Advanced Computer Science and Applications* 8, 10 (2017). https://doi.org/10.14569/IJACSA.2017.081034

[102] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. 2018. Albis: High-Performance File Format for Big Data Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 615–630. https://www.usenix.org/conference/atc18/presentation/trivedi

[103] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 307–320.

[104] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. 2015. A Tale of Two Erasure Codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 213–226. https://www.usenix.org/conference/fast15/technical-sessions/presentation/xia

[105] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2101–2113. https://doi.org/10.14778/3476249.3476265

[106] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2023. Enhancing Computation Pushdown for Cloud OLAP Databases. *https://arxiv.org/pdf/2312.15405.pdf* (2023).

[107] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. 2021. Ship Compute or Ship Data? Why Not Both?. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 633–651. https://www.usenix.org/conference/nsdi21/presentation/you

[108] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1802–1805. https://doi.org/10.1109/ICDE48307.2020.00174

[109] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[110] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (oct 2023), 148–161. https://doi.org/10.14778/3626292.3626298