FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS

Yifei Yang[1], Matt Youill[2], Matthew Woicik[3], Yizhou Liu[1],
Xiangyao Yu[1], Marco Serafini[4], Ashraf Aboulnaga[5], Michael Stonebraker[3]
[1]University of Wisconsin-Madison, [2]Burnian, [3]Massachusetts Institute of Technology, [4]University of Massachusetts-Amherst, [5]Qatar Computing Research Institute
[1]{yyang673@, liu773@, yxy@cs.}wisc.edu, [2]matt.youill@burnian.com, [3]{mwoicik@, stonebraker@csail.}mit.edu,
[4]marco@cs.umass.edu, [5]aaboulnaga@hbku.edu.qa

ABSTRACT

Modern cloud databases adopt a *storage-disaggregation* architecture that separates the management of computation and storage. A major bottleneck in such an architecture is the network connecting the computation and storage layers. Two solutions have been explored to mitigate the bottleneck: *caching* and *computation pushdown*. While both techniques can significantly reduce network traffic, existing DBMSs consider them as orthogonal techniques and support only one or the other, leaving potential performance benefits unexploited.

In this paper we present *FlexPushdownDB* (*FPDB*), an OLAP cloud DBMS prototype that supports fine-grained hybrid query execution to combine the benefits of caching and computation pushdown in a storage-disaggregation architecture. We build a hybrid query executor based on a new concept called *separable operators* to combine the data from the cache and results from the pushdown processing. We also propose a novel *Weighted-LFU* cache replacement policy that takes into account the cost of pushdown computation. Our experimental evaluation on the Star Schema Benchmark shows that the hybrid execution outperforms both the conventional *caching-only* architecture and *pushdown-only* architecture by 2.2×. In the hybrid architecture, our experiments show that Weighted-LFU can outperform the baseline LFU by 37%.

PVLDB Reference Format:
Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, Michael Stonebraker. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. PVLDB, 14(11): 2101 - 2113, 2021.
doi:10.14778/3476249.3476265

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/cloud-olap/FlexPushdownDB.git.

1 INTRODUCTION

Database management systems (DBMSs) are gradually moving from on-premises to the cloud for higher elasticity and lower cost. Modern cloud DBMSs adopt a *storage-disaggregation architecture* that divides computation and storage into separate layers of servers connected through the network, simplifying provisioning and enabling independent scaling of resources. However, disaggregation requires rethinking a fundamental principle of distributed DBMSs: "move computation to data rather than data to computation". Compared to the traditional shared-nothing architecture, which embodies that principle and stores data on local disks, the network in the disaggregation architecture typically has lower bandwidth than local disks, making it a potential performance bottleneck.

Two solutions have been explored to mitigate this network bottleneck: *caching* and *computation pushdown*. Both solutions can reduce the amount of data transferred between the two layers. Caching keeps the hot data in the computation layer. Examples include Snowflake [21, 48] and Presto with Alluxio cache service [14]. The Redshift [30] layer in Redshift Spectrum [8] can also be considered as a cache with user-controlled contents. With computation pushdown, filtering and aggregation are performed close to the storage with only the results returned. Examples include Oracle Exadata [49], IBM Netezza [23], AWS Redshift Spectrum [8], AWS Aqua [12], and PushdownDB [53]. The fundamental reasons that caching and pushdown have performance benefits are that local memory and storage have higher bandwidth than the network and that the internal bandwidth within the storage layer is also higher than that of the network.

Existing DBMSs consider caching and computation pushdown as *orthogonal*. Most systems implement only one of them. Some systems, such as Exadata [49], Netezza [23], Redshift Spectrum [8], and Presto [14] consider the two techniques as independent: query operators can either access cached data (i.e., full tables) or push down computation on remote data, but not both.

In this paper, we argue that caching and computation pushdown are *not* orthogonal techniques, and that the rigid dichotomy of existing systems leaves potential performance benefits unexploited. We propose *FlexPushdownDB* (*FPDB* in short), an OLAP cloud DBMS prototype that combines the benefits of caching and pushdown.

FPDB introduces the concept of *separable operators*, which combine local computation on cached segments and pushdown on the segments in the cloud storage. This hybrid execution can leverage cached data at a fine granularity. While not all relational operators are separable, some of the most commonly-used ones are, including filtering, projection, aggregation. We introduce a *merge* operator to combine the outputs from caching and pushdown.

Separable operators open up new possibilities for caching. Traditional cache replacement policies assume that each miss requires

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476265

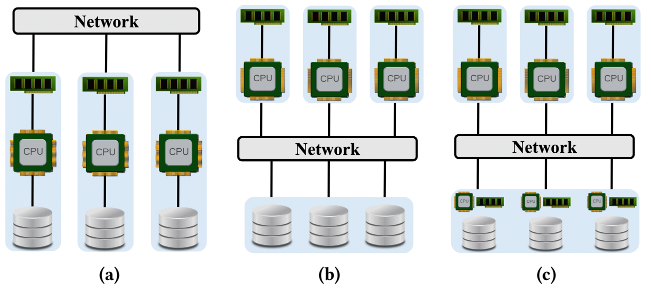


Figure 1: Distributed Database Architectures — (a) shared-nothing, (b) shared-disk, and (c) storage-disaggregation.

loading the missing data block to the cache, which incurs a constant cost if the blocks have the same size. In *FPDB*, however, this assumption is no longer true because we can push down computation instead of loading data. The cost of a miss now depends on how amenable the block is to pushdown—for misses that can be accelerated with pushdown (e.g., high-selectivity filters), the cost of a miss is lower. We develop a new benefit-based caching framework and a new caching policy called *Weighted-LFU*, which incorporates caching and pushdown into a unified cost model to predict the best cache eviction decisions. We compare Weighted-LFU with popular conventional policies, including *least-recently used* (LRU), *least-frequently used* (LFU), and an optimal policy called *Belady* [18] that assumes the availability of future access patterns. Our evaluation shows that *Weighted-LFU* outperforms all these policies.

In summary, the paper makes the following key contributions:

- We develop a fine-grained hybrid execution mode for cloud DBMSs to combine the benefits of caching and pushdown in a storage-disaggregation architecture. The hybrid mode outperforms both *Caching-only* and *Pushdown-only* architectures by 2.2× on the Star Schema Benchmark (SSB) [38].
- We study the performance effect of various caching management policies. We developed a novel *Weighted-LFU* replacement policy that is specifically optimized for the disaggregation architecture. *Weighted-LFU* outperforms the second-best policy (i.e., LFU) by 37% and Belady by 47%.
- We present the detailed design and implementation of *FPDB*, an open-source C++-based cloud-native OLAP DBMS. We believe it can benefit the community given the lack of cloud-DBMS prototypes.

The rest of the paper is organized as follows. Section 2 presents the background and motivation of our design. After showing an overview of *FPDB* in Section 3, Sections 4, 5, and 6 discuss the hybrid execution model, the caching policy, and the detailed implementation of *FPDB*, respectively. Section 7 evaluates the performance of *FPDB* over baseline architectures and policies. Finally, Section 8 discusses related work and Section 9 concludes the paper.

2 BACKGROUND AND MOTIVATION

This section describes the background on the storage-disaggregation architecture (Section 2.1) and the new challenges, as well as why existing systems are sub-optimal in solving the problem (Section 2.2).

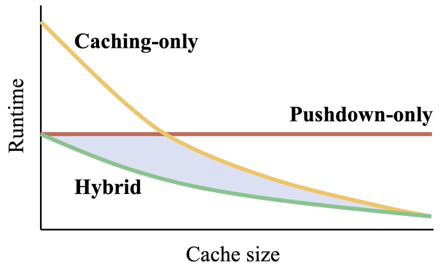


Figure 2: Performance trade-off between caching, computation pushdown, and an ideal hybrid approach.

2.1 Storage-disaggregation Architecture

According to the conventional wisdom, *shared-nothing* (Figure 1(a)) is the best architecture for high-performance distributed data warehousing systems, where servers with local memory and disks are connected through a network. While a cloud DBMS can also adopt a shared-nothing architecture, many cloud-native databases choose to disaggregate the computation and storage layers (Figure 1(c)). This brings benefits of lower cost, simpler fault tolerance, and higher hardware utilization. Many cloud-native databases have been developed recently following such an architecture, including Aurora [46, 47], Redshift Spectrum [8], Athena [7], Presto [11], Hive [43], SparkSQL [16], Snowflake [21], and Vertica EON mode [36, 45].

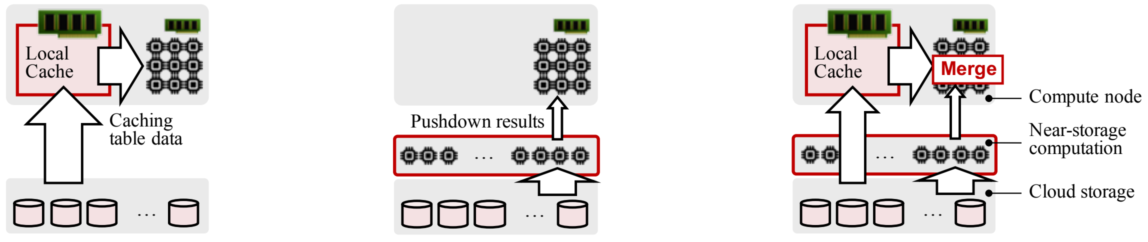
Besides the separation of computation and storage, the disaggregation architecture also supports *limited computation within the storage layer.* The computation can happen either on the storage nodes (e.g., Aurora [46]) or in a different layer close to the storage nodes (e.g., Redshift Spectrum [8] and Aqua [12]). This allows a database to push filtering and simple aggregation operations into the storage layer, leading to performance improvement and potential cost reduction as shown in previous work [53].

While the storage-disaggregation architecture sounds similar to the well-known *shared-disk* architecture (Figure 1(b)), they actually have significant differences. In a shared-disk architecture, the disks are typically centralized, making it hard to scale out the system. The disaggregation architecture, by contrast, can scale the storage layer horizontally (both the computation and the capacity) just like the computation layer. The disaggregation architecture also provides non-trivial computation in the storage layer, while disks are passive storage devices in the shared-disk architecture.

2.2 Challenges in Disaggregated DBMSs

The network connecting the computation and storage layers is a major performance bottleneck in the storage-disaggregation architecture. The network bandwidth is significantly lower than the disk IO bandwidth within a single server, leading to lower performance than a shared-nothing DBMS [42]. Existing cloud DBMSs use two solutions to alleviate this problem: *caching* and *computation pushdown.* Figure 2 shows the high-level performance trade-off between the two approaches, which we describe below.

Solution 1: Caching. These systems keep hot data in the local memory or disks of the computation nodes. The high-level architecture is shown in Figure 3(a). Cache hits require only local data



(a) Caching Only (b) Pushdown Only (c) *FPDB*

Figure 3: System Architectures — High level architectural comparison between *Caching-only*, *Pushdown-only*, and *FPDB*.

accesses and are thus much faster than cache misses, which require loading data over the network. As shown in Figure 2, as the cache size increases, the query execution time decreases due to a higher hit ratio. Snowflake [21] and Presto [14] support caching with the LRU replacement policy. Redshift Spectrum [8] does not support dynamic caching but allows users to specify what tables to keep in the Redshift layer for fast IO. We are not aware of any thorough study of more advanced caching policies in these systems.

Solution 2: Computation pushdown. The idea here is to perform certain DBMS operators close to where the data resides; the high-level architecture is shown in Figure 3(b). Computation pushdown has been widely explored in the context of database machines [23, 25, 44, 49], Smart Disk/SSD [22, 26, 29, 35, 51, 52], processing-in-memory (PIM) [27, 34], and cloud databases [8, 12, 37, 53]. They are all based on the observation that the internal bandwidth in the storage is higher than the external bandwidth between storage and computation and that pushdown can reduce data transfer; the problem is particularly severe in cloud databases due to disaggregation. As shown in Figure 2, the performance of computation pushdown is independent of the cache size—the DBMS does not exploit caching.

When the cache size is small, pushdown outperforms caching due to reduced network traffic; when the cache size is sufficiently large, caching performs better due to a higher cache hit ratio. Ideally, a system should adopt a hybrid design that combines the benefits of both worlds—caching a subset of hot data and pushdown computation for the rest, which is shown as the bottom line in Figure 2.

Existing systems do not offer a fully hybrid design. While some systems support both caching and pushdown, they select the operation mode at the table granularity. The storage layer keeps additional "external" tables that can be queried using computation pushdown. But no system, to the best of our knowledge, can dynamically cache external tables at a fine granularity (e.g., subsets of columns) without an explicit load command. A key challenge here is to split a general query to exploit both techniques at a fine granularity. Another key challenge is to design a caching policy that is aware of computation pushdown. *FPDB* addresses both challenges.

3 *FPDB* OVERVIEW

3.1 System Architecture

Figure 3(c) shows the high-level architecture of *FPDB* in contrast to existing architectures of *Caching-only* (Figure 3(a)) and *Pushdown-only* (Figure 3(b)). Similar to *Caching-only*, *FPDB* stores the hot input data in the local cache (i.e., main memory or disk) to take advantage of fast IO. Similar to *Pushdown-only*, *FPDB* keeps the cold input data in the external cloud storage and pushes down filters to reduce network traffic. *FPDB* contains the following two main modules to enable such hybrid query execution:

Hybrid query executor. The query executor takes in a logical query plan from the optimizer and transforms it into a *separable query plan* based on the content in the cache. The separable query plan processes the cached data in the compute node and pushes down filters and aggregation to the storage layer for uncached data. The two portions are then merged and fed to downstream operators. Section 4 will discuss the details of how operators are separated and merged.

Cache manager. The cache manager determines what data should be cached in the computation node. The cache eviction policy takes into account the existence of computation pushdown to exploit further performance improvement. For each query, the cache manager updates the metadata (e.g., access frequency) for the accessed data and determines whether admission and/or eviction should occur.

3.2 Design Choices

Designing *FPDB* requires making two high-level design decisions: *what to cache* and *at which granularity*, which we discuss below.

Caching Table Data or Query Results. Two types of data can potentially be cached in *FPDB*: *table data* and *query results*. Table data can be either the raw input files or a subset of rows/columns of the input tables. Query results can be either the final or intermediate results of a query, which can be considered as materialized views.

We consider the caching of table data and results as two orthogonal techniques, with their own opportunities and challenges. In *FPDB*, we explore caching only the table data since it is a simpler problem. The extension to result caching is left to future work.

Storage and Caching Granularity. *FPDB* stores tables in a distributed cloud storage service. Tables are horizontally partitioned based on certain attributes (e.g., primary key, sorted field, timestamp, etc.). Each *partition* is stored as a file in the cloud storage and contains all the columns for the corresponding subset of rows.

The basic caching unit in *FPDB* is a *segment*, which contains data for a particular column in a table partition (i.e., a column for a subset of rows). A segment is uniquely identified by the *segment key*, which contains three parts: the table name, the partition name, and the column name. The data format of a segment (e.g., Apache

Arrow) can be potentially different from the data format of the raw table file (e.g., CSV or Parquet).

4 HYBRID QUERY EXECUTOR

At a high level, the query executor converts the logical query plan into a *separable query plan* by dispatching *separable operators* into both local and pushdown processing; the results are then combined through a *merge operator*. This section describes these modules and then illustrates how the system works through an example query.

4.1 Separable Operators

We call an operator *separable* if it can be executed using segments in both the cache and the cloud storage, with the results combined as the final output. Not all the operators are separable (e.g., a join). Below we analyze the separability of several common operators.

Projection. Projection is a separable operator. If only a subset of segments in the queried columns are cached, the executor can load the remaining segments from the storage layer. The results can be then combined and fed to the downstream operator.

Filtering scan. Whether a filtering scan is separable depends on the cache contents. Ideally, the executor can process some partitions in the cache, push down filtering for the remaining partitions to the storage, and then merge the results, thus separating the execution. However, the situation can be complex when multiple columns are selected but not all of them are part of the filtering predicate.

Consider a scan query that returns two sets of attributes A and B from a table but the filtering predicate is applied on only attribute set A. For a particular table partition, if all segments in both A and B are cached, the partition can be processed using the data in the cache. However, if only a subset of segments in A are cached, the executor must either load the missing segments or push down the scan of the partition to the storage. Finally, if all segments in A but only a subset of segments in B are cached (call it B'), the processing can be partially separated—the executor filters A and B' using cached data, and pushes down the filtering for $(B - B')$.

Base table aggregation. Pushdown computation can perform aggregation on certain columns of a table. These operators can be naturally separated: a partition is aggregated locally if all involved segments are cached; otherwise the aggregation is pushed down to the storage. The output can then be merged. For aggregation functions like `average`, both local and remote sides return the `sum` and `count` based on which the average is calculated.

Hash join. A join cannot be completely pushed down to the storage layer due to limitations of the computation model that a storage layer supports. Prior work [53] has shown that a bloom hash join can be partially pushed down as a regular predicate on the outer relation in a join. Given this observation, we conclude that the *building phase* in hash join is not separable—the columns of interest in the inner relation must be loaded to the compute node. The *probing phase* is separable: cached segments of the outer relation can be processed locally, while uncached segments can be filtered using the bloom filter generated based on the inner relation.

Sort. Theoretically, sort is separable—a remote segment can be sorted through pushdown and the segments are then merged in the computation node. Such techniques have been explored in the context of near-storage processing [32] using FPGA. However, since the cloud storage today does not support sorting (e.g., S3 Select), the separation of sorting is not supported in *FPDB*.

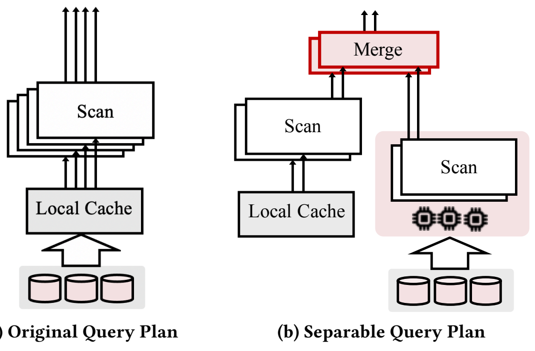


Figure 4: Example of a Separable Query Plan — The hybrid query plan contains a parallel *merge* operator that combines the results from cache and computation pushdown.

4.2 Separable Query Plan

A query plan is separable if it contains separable operators. Figure 4 shows examples of a conventional query plan without pushdown (Figure 4(a)) and a separable query plan (Figure 4(b)).

A conventional query plan reads all the data from the computation node's local cache (i.e., buffer pool). For a miss, the data is loaded from the storage layer into the cache before query processing. A separable query plan, by contrast, splits its *separable operators* and processes them using both the cached data and pushdown computation. How the separation occurs depends on the current content in the cache, as described in Section 4.1.

For good performance and scalability, the merge operator in *FPDB* is implemented across multiple parallel threads. Specifically, each operator in *FPDB* is implemented using multiple worker threads and each worker thread is assigned multiple segments of data. The segments assigned to a particular worker might be entirely cached, entirely remote, or a mixture of both. For threads with a mixture of data sources, the results must be first merged locally into a unified data structure. The data across different threads does not need to be explicitly merged—they are directly forwarded to the downstream operators (e.g., joins) following the original parallel query plan.

4.3 Example Query Execution

```
SELECT R.B, sum(S.D)
FROM R, S
WHERE R.A = S.C AND R.B > 10 AND S.D > 20
GROUP BY R.B
```

Listing 1: Example query joining relations R and S.

We use the query above as an example to further demonstrate how the hybrid query executor works; the plan of the query is shown in Figure 5. The example database contains two relations R and S with the assumption that $|R| < |S|$, and each relation has two partitions (as shown in the cloud storage in Figure 5). Relation R has two attributes A and B, and relation S has two attributes C and

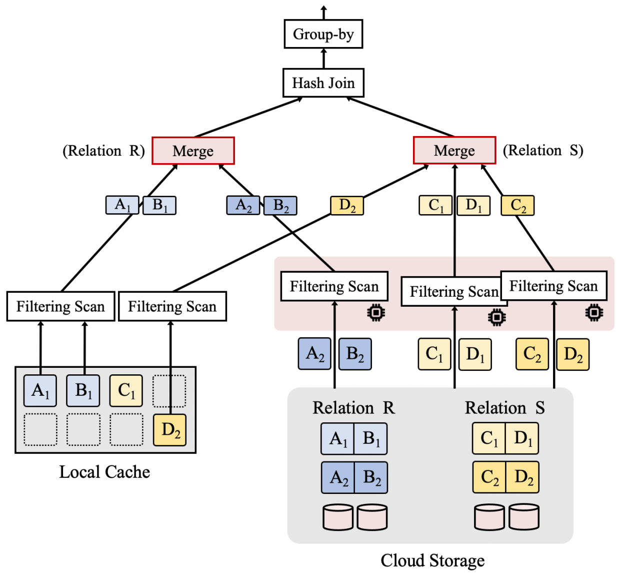


Figure 5: Separable Query Plan — For the query in Listing 1.

D. Four segments are cached locally, as shown in the Local Cache module in Figure 5.

To execute the query using hash join, the DBMS first scans *R* to build the hash table and scans *S* to probe the hash table. The output of the join is fed to the group-by operator. Four partitions are involved in the join, i.e., partitions 1 and 2 in relations *R* and *S*, respectively. Depending on what segments are cached, the partition can be scanned locally, remotely, or in a hybrid mode.

Scanning relation *R*. For the first partition in *R*, both segments (i.e., A_1 and B_1) are cached. Therefore, the executor reads them from the local cache and no pushdown is involved. For the second partition in *R*, neither segment (i.e., A_2 and B_2) is cached, thus the filter is pushed down to the storage layer, which returns the subset of rows in A_2 and B_2 that satisfy the predicate. Finally, the local and remote results are combined through a merge operator.

Scanning relation *S*. For the first partition in *S*, only segment C_1 is cached, but the filter predicate of relation *S* is on attribute *D*, so the filtering scan cannot be processed locally and must be pushed down to the storage, which returns the filtered segments C_1 and D_1. For the second partition in *S*, only segment D_2 is cached. Since the filter predicate is on *D*, the DBMS can directly read from the cache to process D_2. Since the scan should also return attribute C_2, the DBMS can push down the filter to the storage to load C_2. Note that it is also possible to process this partition by pushing down the processing of both segments C_2 and D_2, namely, ignoring the cached D_2. This alternative design avoids evaluating the predicate twice (i.e., for the cached data and remote data) but incurs more network traffic. *FPDB* adopts the former option.

In the discussion so far, only the filtering scan is executed in a hybrid mode. As described in Section 4.1, the probe table in a hash join can also be partially pushed down. For the example query in particular, the DBMS can scan relation *R* first, builds a *bloom filter* on attribute *R.A*, and consider this bloom filter as an extra predicate when scanning relation *S*; namely, the predicates on *S* then become `S.D > 20 AND bloom_filter(S.C)`. Note the bloom filter can only be constructed after the entire column of attribute *R.A* is loaded. Therefore, when pushing down the probe table of the hash join, the scan of relation *S* can start only after the scan of relation *R* completes. In contrast, both scans can be executed in parallel when the bloom filter is not pushed down.

4.4 Execution Plan Selection

FPDB currently uses heuristics to generate separable query plans. It takes an initial plan from the query optimizer, and splits the execution of separable operators based on the current cache content. Specifically, an operator on a partition is always processed based on cached segments whenever the accessed data is cached. Otherwise, we try to pushdown the processing of the partition as much as we can. If neither works (e.g., the operator is not separable), we fall back to the pullup solution and load the missing segments from the storage layer. Note that the heuristics we adopt can generate only one separable plan given an input query plan. We adopt these heuristics based on the following two assumptions:

- Local processing on cached data is more efficient than pushdown processing in the storage layer.
- Pushdown processing is more efficient than fetching all the accessed segments and then processing locally.

The two conditions can hold in many cloud setups with storage disaggregation. The computation power within the storage layer is still limited compared to the local computation nodes and the network between the compute and storage layers has lower bandwidth than the aggregated disk IO bandwidth within the storage layer. Evaluation in Section 7 will demonstrate the effectiveness of the heuristics in improving system performance.

Although our heuristics are simple and effective, we do note that they may not always lead to a globally optimal separable query plan, because the two assumptions may not hold universally. For example, pushdown may outperform due to its massive parallelism even if all the data is cached. An important follow-up work is to develop a *pushdown-aware* query optimizer that can deliver better performance than the simple heuristics. We leave such exploration to future work.

5 CACHE MANAGER

The cache manager decides what table segments should be fetched into the cache and what segments should be evicted, as well as when cache replacement should happen. We notice a key architectural difference in *FPDB* that makes conventional cache replacement policies sub-optimal: in a conventional cache-only system, cache misses require loading data from storage to cache. If cached segments are of equal size, each cache miss incurs the same cost. In *FPDB*, however, we can push down computation instead of loading data. Some segments may be more amenable to pushdown than others, which affects the benefit of caching. In other words, segments that cannot be accelerated through pushdown should be considered for caching with higher weight; and segments that can already be significantly accelerated through pushdown can be cached with lower weight—the extra benefit of caching beyond pushdown is relatively smaller.

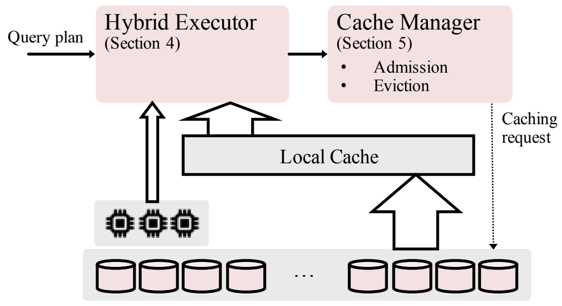


Figure 6: Integration of Executor and Cache Manager — The cache manager determines what segments should stay in cache.

We develop a *Weighted-LFU (WLFU)* cache replacement policy for *FPDB* based on this observation.

In the rest of this section, we describe the integration of the cache manager and the hybrid executor in Section 5.1 and present a generic *benefit-based caching framework* in Section 5.2. We describe how conventional replacement policies (i.e., LRU, LFU, and Belady) fit into this framework in Section 5.3 and present the proposed Weighted-LFU policy in Section 5.4.

5.1 Integration with Hybrid Executor

Figure 6 demonstrates how the cache manager is integrated with the hybrid executor in *FPDB*. The hybrid executor takes a query plan as input and sends information about the accessed segments to the cache manager. The cache manager updates its local data structures, determines which segments should be admitted or evicted, and loads segments from the cloud storage into the cache.

For cache hits, the hybrid executor processes the query using the cached segments. Cache misses include two cases: First, if the caching policy decides *not* to load the segment into the cache, then *FPDB* exploits computation pushdown to process the segment. Otherwise, if the caching policy decides to cache the segment, the DBMS can either wait for the cache load or push down the computation. The tradeoff here is between query latency and network traffic. *FPDB* adopts the former to minimize network traffic.

5.2 Benefit-Based Caching Framework

The cache manager determines which segments to admit or evict. The key design decisions here are the *admission* and *eviction* policies, which vary depending on the algorithm being implemented.

Ideally, a cache should contain the segments that can benefit the most from caching. We use the benefit value, *s.benefit*, to represent the benefit that segment *s* can get from staying in the cache. The key differences among caching policies are the different ways of determining the benefit value. For example, the benefit value represents recency and frequency information for LRU and LFU policies, respectively. The details on determining the benefit value will be discussed in Sections 5.3 and 5.4.

Algorithm 1 shows how the benefit-based caching framework works in *FPDB*. When a set of segments are accessed, their benefit values are updated (lines 1–2) to reflect the new accesses. Then for each missing segment *s* in the order of decreasing benefit, we try to cache the segment if there is still space in the cache (lines 3–6). If the space is insufficient, we make space by evicting the segment with the least benefit (i.e., segment t) from the cache until t has a higher benefit value than the newly updated segments (lines 8–12), at which point the caching algorithm stops loading new segments.

Algorithm 1: Benefit-Based Caching Framework

```
   Input: list of accessed segments $S_{in}$
1  foreach $s$ in $S_{in}$ do
2      update the benefit of $s$
3  while $s \leftarrow$ get missing segment in $S_{in}$ with highest benefit do
4      if cache has enough space to hold $s$ then
5          issue request to load segment $s$
6          $S_{in}$.remove($s$)
7      else
8          $t \leftarrow$ get segment in cache with lowest benefit
9          if $t.benefit < s.benefit$ then
10             evict segment $t$ from cache
11         else
12             break
```

Note that our framework makes a few simplifying assumptions regarding the caching policy. For example, we assume an eager admission policy where a segment is immediately admitted when it can be. The algorithm is also greedy since we consider segments for caching strictly in the decreasing order of benefit values. Alternative designs that break these assumptions should be orthogonal to the rest of the paper and beyond the scope of this work.

5.3 Traditional Replacement Policies

We explore three traditional caching policies: *least-recently-used* (LRU), *least-frequently-used* (LFU), and an optimal policy *Belady*.

Least-Recently Used (LRU). LRU is the most widely used cache replacement policy. A new cache miss always replaces the cached segment(s) that are least-recently used. LRU assumes that more recently used data are also likely to be used again in the future, thus having a higher priority to be cached. In our framework, this means the benefit value of a segment is simply the timestamp of its last access. Segment size does not play a role in our LRU algorithm.

Least-Frequently Used (LFU). In LFU, the *frequency counter* of a segment is incremented by *1/(segment.size)* whenever it is accessed. The counter captures the size-normalized access frequency of the segment. If a missing segment's counter is larger than the smallest counter of cached segments, the least-frequently used segment in the cache is replaced. In our framework, the benefit value of segment *s* is simply the size-normalized frequency counter.

Belady Replacement Policy. Belady [18] is an optimal replacement policy that assumes availability of future information. Specifically, upon a miss, the algorithm replaces the segment that will not be accessed again for the longest time in the future. The benefit value of a segment *s* is therefore *1/(the number of queries until the segment is accessed again)*.

The Belady algorithm has been proven optimal when all the segments have the same size and all cache misses have the same cost. Since these assumptions are not always true in *FPDB* (due to variable segment size and pushdown), Belady is no longer optimal in our setting. Even so, we use it as a baseline to gain insights about different replacement policies.

5.4 Weighted-LFU Replacement Policy

As discussed at the beginning of this section, the hybrid caching and pushdown design in *FPDB* changes a fundamental assumption of cache replacement—cache misses for different segments incur different costs. Specifically, consider two segments, *A* and *B*, where *A* is accessed slightly more frequently than *B*, so that an LFU policy prefers caching *A*. However, it can be the case that segment *A* can benefit from computation pushdown so that a cache miss is not very expensive, while segment *B* is always accessed with no predicate hence cannot benefit from pushdown. In this case, it might be more beneficial if the DBMS prefers *B* over *A* when considering caching.

In the benefit-based caching framework, we can incorporate the above intuition by taking pushdown into consideration when calculating the benefit values. Specifically, we modify the existing LFU policy to achieve this goal. Instead of incrementing the frequency counter by *1/segment.size* for each access to a segment, we increment the counter by a *weight*, the value of which depends on whether the segment can be pushed down and if so, what cost pushdown is. Intuitively, the more costly the pushdown is, the more benefit we get from caching, hence the higher weight.

While there are many different ways to calculate the weight, in *FPDB*, we choose a straightforward formulation to represent a weight by the estimated total amount of work (measured in time) of pushdown computation, which is modeled by three components: time of network transfer, time of data scanning, and time of computation, as shown in Equation 1. The total time is divided by the segment size to indicate the size-normalized benefit of caching.

$$w(s) = \frac{total\_work(s)}{size(s)} = \frac{t_{net}(s) + t_{scan}(s) + t_{compute}(s)}{size(s)} \quad (1)$$

We estimate the time of each component using the following simple equations (Equations 2–4).

$$t_{net}(s) = \frac{selectivity(s) \times size(s)}{BW_{net}} \quad (2)$$

$$t_{scan}(s) = \frac{N_{tuples}(s) \times size(tuple)}{BW_{scan}} \quad (3)$$

$$t_{compute}(s) = \frac{N_{tuples}(s) \times N_{predicates}}{BW_{compute}} \quad (4)$$

The equations above assume the data within the cloud storage is in row-oriented format (e.g., CSV)—they can be easily accommodated for column-format data (e.g., Parquet) by adjusting data scan amount to the size of columns accessed instead of the whole object. The time of each component is essentially the total amount of data transfer or computation divided by the corresponding processing bandwidth. Most of the parameters in the numerators can be statically determined from the statistics (e.g., $size(s)$, $size(tuple)$, $N_{tuples}(s)$) or from the query (e.g., $N_{predicates}$), except for $selectivity(s)$ which can be derived after the corresponding segment has been processed by the executor.

For the bandwidth numbers in the denominators, we run simple synthetic queries that exercise the corresponding components to estimate their values. This process is performed only once before all the experiments are conducted.

6 IMPLEMENTATION

Given that not many open-source cloud-native DBMSs exist, we decided to implement a new prototype, *FPDB*, in C++ and make the code publicly available to the community[1]. In this section, we describe different aspects of our system.

6.1 Cloud Environment

FPDB adopts a storage-disaggregation architecture. We choose the widely used AWS Simple Storage Service (S3) [9] as the cloud storage service. We also use S3 Select [31], a feature where limited computation can be pushed down onto S3, including projection, filtering scan, and base table aggregation. *FPDB* currently supports a single computation node.

FPDB requests data from S3 through AWS C++ SDK. We configure rate limits, timeouts, and connections in AWS *ClientConfiguration* high enough for better performance. Besides, *FPDB* does not use HTTPS/SSL which incurs extra overhead, as we expect analytics workloads would typically be run in a secure environment.

6.2 Data Format

FPDB supports table data in the cloud storage in both CSV and Parquet [4] format. Within the database engine and the cache, we use Apache Arrow [3] to manage data; table data is converted to Arrow within the scan operator. Arrow is a language-agnostic columnar data format designed for in-memory data processing. In the executor, we encapsulated Arrow's central type, the *Array*, to represent a data segment. Arrays are one or more contiguous memory blocks holding elements of a particular type. The number of blocks required depends on several parameters such as the element type, whether null values are permitted, and so on. Using the same data format for the processing engine and the cache eliminates the overhead of an extra data type conversion.

6.3 In-Memory Parallel Processing

Below we describe the details of *FPDB*'s in-memory parallel processing engine in a bottom-up fashion.

Expression Evaluation. *FPDB* uses Gandiva [10] for efficient expression evaluation. Gandiva compiles and evaluates expressions on data in Arrow format. It uses LLVM to perform vectorized processing and just-in-time compilation to accelerate expression evaluation and exploit CPU SIMD instructions for parallelism.

While Gandiva supports multi-threaded expression *execution* we found multi-threaded expression *compilation* troublesome. An attempt to compile multiple expressions simultaneously often failed in LLVM code generation or produced incomplete expressions. Therefore, we use serial compilation only.

Actor-Based Parallel Processing. *FPDB* supports parallel query execution using the C++ Actor Framework (CAF) [19]. CAF is a lightweight and efficient implementation of the Actor model [15] similar to those found in Erlang [17] or Akka [1]. Queries are composed of a number of operator types arranged in a tree. Scan

[1]https://github.com/cloud-olap/FlexPushdownDB.git

operators form the leaves and a single collate operator forms the root. Operators communicate via message passing from producers to consumers, i.e., messages flow from leaves to the root.

Each operator type is instantiated into multiple actors. For the scan operator, an actor is created for each input table partition. For other operators, the number of actors is determined by the number of CPU cores. By default, CAF multiplexes the execution of these actors over all CPU cores on the host machine.

When query execution starts, a query coordinator instructs each operator to begin execution, in parallel, and then tracks each operator's completion status. Operators perform their work, which may depend on the work of upstream operators, and send output messages to consumers, in a pipelined fashion. On completion of all operators the coordinator delivers the query result, from the collate operator, to the client executing the query.

Finally, both parallel hash join and parallel aggregation are implemented in *FPDB*. With actor-based parallel processing, *FPDB* can achieve a CPU utilization of more than 96%.

Caching. We perform caching using the main memory (RAM) of the compute nodes. The cache in *FPDB* is also implemented as an actor with cache contents managed by the caching policy. The cache communicates with other operators through message passing to admit and evict data segments.

7 EVALUATION

In this section, we evaluate the performance of *FPDB* by focusing on the following key questions:

- How does the hybrid architecture perform compared to baseline architectures?
- How does Weighted-LFU perform compared to traditional cache replacement policies?
- What is the impact of the network bandwidth?
- What is the resource usage and monetary cost of FPDB?

7.1 Experimental Setup

Server configuration. We conduct all the experiments on AWS EC2 compute-optimized instances. To thoroughly evaluate the performance, we pick two instance types with different network bandwidth, c5a.8xlarge (which costs \$1.52 per hour in US-West-1 pricing), with 32 vCPU, 64 GB memory, and a 10 Gbps network bandwidth, and c5n.9xlarge (which costs \$2.43 per hour in US-West-1 pricing), with 36 vCPU, 96 GB memory, and a 50 Gbps network bandwidth. Unless otherwise specified, we use the c5a.8xlarge instance as default. The server runs the Ubuntu 20.04 operating system.

Benchmark. We use the Star Schema Benchmark (SSB) [38] with a scale factor 100. Each table is partitioned into objects of roughly 150 MB when using CSV format, which is large enough to avoid generating many scan operators and the associated scheduling overhead. Each request to S3 is responsible for 15 MB data, which is small enough to leverage parallelism when loading data from the storage layer. Parquet data is derived from CSV data. The SSB dataset has five tables—four *dimension tables* and one *fact table*. The fact table is much larger than the dimension tables, and contains foreign keys which refer to primary keys in the dimension tables.

We implement a *random query generator* based on SSB queries. A query is generated based on a *query template* with parameters in the filter predicates randomly picked from a specified range of values (or a set of values for categorical data). We incorporate *skewness* into the benchmark by picking the values following a *Zipfian* [28] distribution with tunable skewness that is controlled by a parameter θ. Skewness is applied to the fact table (i.e. *lineorder*) such that more recent records are accessed more frequently. A larger θ indicates higher skewness. The default value of θ is 2.0, where about 80% queries access the 20% most recent data (i.e. hot data). We set the default cache size to 8 GB which is enough to cache the hot data under the default skewness.

Measurement. Each experiment executes a batch of queries sequentially. The experiment contains a *warmup phase* and an *execution phase*. The warmup phase contains 50 queries sequentially executed; we have conducted separate experiments and found that 50 is sufficient to warm up the cache. The execution phase contains 50 sequentially executed queries on which we report performance.

Architectures for Comparison. The following system architectures are implemented in *FPDB* for performance comparison:

- *Pullup*: Data is never cached in the computation node. All accesses load table data from S3. This design is used in Hive [43], SparkSQL [16], and Presto [11].
- *Caching-only*: Data is cached in the computation node with no pushdown supported. This design has been adopted in Snowflake [21] and Presto with Alluxio caching [14].
- *Pushdown-only*: Filtering scan is always pushed down to the storage layer whenever possible. Data is never cached in the computation node. This design is used in PushdownDB [53] and Presto with S3 Select enabled.
- *Hybrid*: The hybrid caching and pushdown architecture proposed in this paper.

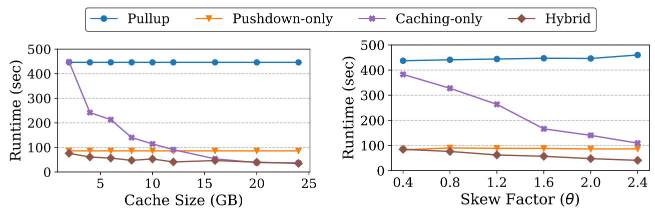
7.2 Performance Evaluation

This subsection evaluates the performance of *FPDB*. We compare and analyze the query execution time of different caching and pushdown architectures. Then we evaluate the effectiveness of the Weighted-LFU caching policy. Finally, we study the impact of network bandwidth between the compute and storage layers.

7.2.1 Caching and Pushdown Architectures.

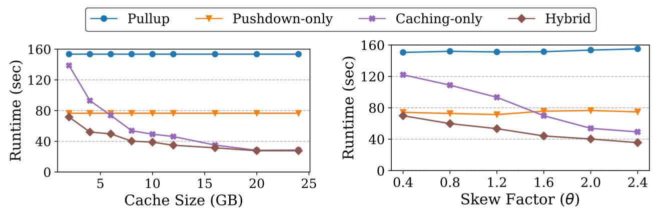
We start with comparing the performance of the hybrid architecture with baseline architectures. We use LFU as the default cache replacement policy for *Caching-only* and *Hybrid*. We report performance varying two parameters: the cache size and the access skewness in the workload (i.e., θ), on both c5a.8xlarge and c5n.9xlarge instances.

Overall Performance. Figure 7 shows the runtime comparison on the c5a.8xlarge instance. Figure 7(a) compares the performance between different caching/pushdown architectures when the cache size changes. First, we observe that the performance of *Pullup* and *Pushdown-only* are not affected by the cache size; this is because data is never cached in either architecture. *Pushdown-only* outperforms *Pullup* by 5.2×, because computation pushdown can significantly reduce the network traffic. The *Caching-only* architecture, in contrast, can take advantage of a bigger cache for higher performance. When the cache is small, its performance is close to *Pullup*. When



(a) Runtime with Varying Cache Size (b) Runtime with Varying Skewness

Figure 7: Performance Comparison (c5a.8xlarge) — The runtime with different (a) cache sizes and (b) access skewness.



(a) Runtime with Varying Cache Size (b) Runtime with Varying Skewness

Figure 8: Performance Comparison (c5n.9xlarge) — The runtime with different (a) cache sizes and (b) access skewness.

the cache size is bigger than 12 GB, it outperforms *Pushdown-only* due to the sufficiently high cache hit ratio.

Finally, the performance of *Hybrid* is consistently better than all other baselines. When the cache is small, *FPDB* behaves like *Pushdown-only*; when the cache is large enough to hold the working set, *FPDB* behaves like *Caching-only*. For cache sizes in between, *Hybrid* can exploit both caching and pushdown to achieve the best of both worlds. At the default cache size (i.e., 8 GB), *Hybrid* outperforms *Pushdown-only* by 80%, *Caching-only* by 3.0×, and *Pullup* by 9.4×. At the crossing point of *Pushdown-only* and *Caching-only* (i.e., roughly 12 GB), *Hybrid* outperforms both by 2.2×.

Figure 7(b) shows the performance of different architectures as the access skewness increases. *Pullup* and *Pushdown-only* are not sensitive to changing skewness. Both *Caching-only* and *Hybrid* see improving performance for higher skew, due to a higher cache hit ratio since there is less hot data.

We further evaluate *FPDB* on the c5n.9xlarge instance which has a 50 Gbps network. Figure 8(a) compares the performance of different architectures with different cache sizes. The general trends are similar to c5a.8xlarge, and *Hybrid* consistently outperforms all baselines. With a higher network bandwidth, *Pushdown-only* is 2× faster than *Pullup*, which is lower than the speedup on the c5a.8xlarge instance because loading segments missing from the cache is faster. The crossing point of *Pushdown-only* and *Caching-only* shifts towards the left to roughly 6 GB, at which point *Hybrid* outperforms both baselines by 51%.

The performance results with increasing access skew on the c5n.9xlarge instance are shown in Figure 8(b). The general trend is also similar to c5a.8xlarge.

From the results above, we observe that different hardware configurations can shift the relative performance of pushdown and caching, but the hybrid design always outperforms both baselines.

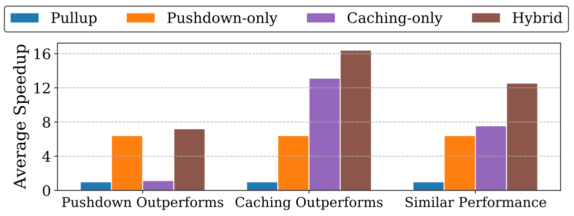


Figure 9: Per-Query Speedup — The average speedup of each representative case in different architectures.

Per-Query Analysis.

We dive deeper into the behavior of the system by inspecting the behavior of individual queries, and observe that they can be categorized into three representative cases: (1) caching has better performance, (2) pushdown has better performance, and (3) both have similar performance. For each category, we compute the average speedup of different architectures compared to *Pullup*. The results on c5a.8xlarge are shown in Figure 9. Although not shown here, the results on c5n.9xlarge have a similar trend.

In all three cases, the performance of *Hybrid* can match the best of the three baseline architectures. When pushdown (or caching) achieves a higher speedup, *Hybrid* slightly outperforms *Pushdown-only* (or *Caching-only*). When the two techniques have similar performance, *Hybrid* can outperform either baseline significantly. The performance results in Figure 7 and Figure 8 are an aggregated effect of these three categories of queries.

Comparison against Existing Solutions.

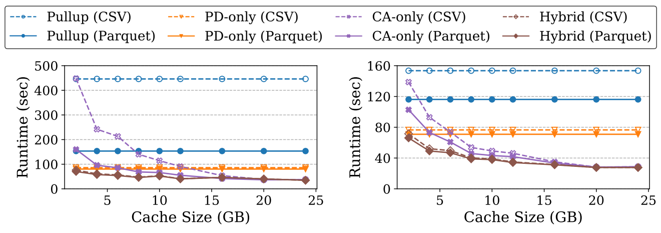
To further validate the performance of our system, we compare *FPDB* with Presto, a production cloud database. We use Presto v0.240 which supports computation pushdown through S3 Select and caching through Alluxio cache service [14]. For Alluxio, we cache data in main memory, which is consistent with *FPDB*. We conduct the experiment under the default workload on an m5a.8xlarge instance[2] on CSV data.

Table 1: Performance Comparsion between Presto and FPDB — The runtime (in seconds) of different architectures in both systems (*Pushdown-only* as *PD-only*, *Caching-only* as *CA-only*) on CSV data.

Architecture	Pullup	PD-only	CA-only	Hybrid
Presto	588.7	271.3	536.3	-
FPDB	472.1	111.2	225.7	80.8

The result is shown in Table 1. *FPDB* outperforms Presto by 25% in *Pullup* and 2.4× in *Pushdown-only*, which implies that query processing inside *FPDB* is efficient. In *Caching-only FPDB* is 2.4× faster than Presto with Alluxio caching. A few reasons explain this performance gain: First, Alluxio caches data at block granularity, which is more coarse-grained than *FPDB*. Second, Alluxio manages cached data through its file system, incurring higher overhead than *FPDB*, which manages cached data directly using heap memory. We

[2]We run Presto on AWS Elastic MapReduce (EMR) which currently does not support c5a and c5n instances in US-West-1. m5a has the most similar configuration.



(a) c5a.8xlarge **(b) c5n.9xlarge**

Figure 10: Parquet Performance — The runtime estimation of different architectures with different cache sizes on Parquet data. Results on CSV data (Figure 7(a), 8(a)) are added for reference.

further note that only *FPDB* supports *Hybrid* query execution. The Alluxio caching layer is unaware of the pushdown capability of S3 while loading data, thus only one technique can be used.

7.2.2 *Parquet Performance.*

In this experiment, we investigate the performance of *FPDB* on data in Parquet [4], a columnar data format commonly used in data analytics. The challenge, however, is that current S3 Select has poor performance on Parquet — pushdown of Parquet processing returns results in CSV, leading to even worse performance than processing CSV data. We studied a few other cloud-storage systems but they either have the same problem [5] or do not support Parquet pushdown at all [2, 13].

In order to estimate the performance of an optimized Parquet pushdown engine, we built an analytical model to predict the system performance under different scenarios [50]. Our model is based on real measurements in software and hardware, and assumes one of the key hardware resources is saturated (e.g., network bandwidth, host processing speed, storage IO bandwidth). In a separate document [50], we present the detailed model and verify that it produces very accurate predictions for CSV data under different cache size, filtering selectivity, among other important configurations. To accurately model Parquet performance, we implement a program that efficiently converts Parquet to Arrow format and processes filtering to mimic the behavior of Parquet pushdown, and plug into the model the performance numbers measured through this program. The key parametric difference between CSV and Parquet includes the amount of network traffic and the speed of pushdown processing.

We perform the estimation for both c5a.8xlarge and c5n.9xlarge instances and report results in Figure 10. We add the performance on CSV data for reference. We observe that the performance on Parquet is always better than the performance on CSV. The gain is more prominent for *Pullup* and *Caching-only* since projection pushdown is free in the Parquet format, leading to network traffic reduction. Gains are less significant for *Pushdown-only* and *Hybrid* since both exploit pushdown already. Comparing Figure 10(a) and 10(b), we also observe that the gain of Parquet is more prominent when the network bandwidth is low, in which case a more severe bottleneck is being addressed.

With Parquet, *Hybrid* still achieves the best performance among all the architectures. When the network is a lesser bottleneck (e.g., faster network or Parquet format), the performance advantage of *Hybrid* is smaller and the crossing point of *Pushdown-only* and *Caching-only* shifts towards the left in the figures. Even with Parquet data and fast network (c5n.9xlarge), at the crossing point, *Hybrid* outperforms both *Pushdown-only* and *Caching-only* by 47%.

We provide the following intuition as to why *Hybrid*'s advantage remains in Parquet data. In essence, the performance gain of pushdown mainly comes from three aspects: (1) network traffic reduction from *projection pushdown*; (2) network traffic reduction from *selection pushdown*; (3) parsing and filtering data with massive parallelism. With Parquet, all architectures have the benefit of (1), but only pushdown processing has the advantages of (2) and (3). For example, under the default cache size (i.e. 8 GB), *Hybrid* reduces network traffic by 66% over *Caching-only* on Parquet data (as opposed to 93% on CSV).

7.2.3 *Weighted-LFU Caching Policy.*

In this experiment we study the cache replacement policy proposed in this paper, Weighted-LFU (cf. Section 5.4). Figure 11 compares it with conventional cache replacement policies. We have conducted a separate experiment which shows that LRU has worse performance than LFU and Belady, and thus exclude it from this experiment.

In the default SSB queries, predicates on different attributes have similar selectivity, making the pushdown cost of different segments similar. To measure the effectiveness of Weighted-LFU, we change the SSB queries to incorporate different pushdown costs, by varying the selectivity of filter predicates. Specifically, we change predicates on some attributes to equality predicates which have very high selectivity (e.g. $lo\_quantity = 10$) while keeping others range predicates (e.g. $lo\_discount < 3$ *or* $lo\_discount > 6$).

As Figure 11 shows, WLFU consistently outperforms the baseline LFU and Belady. The biggest speedup happens when $\theta = 0.3$ (i.e., low access skewness), where WLFU outperforms LFU and Belady by 37% and 47%, respectively. We further measure network traffic incurred and find WLFU reduces network traffic by 66% and 78%, compared to the baseline LFU and Belady respectively. Recall that the optimization goal of WLFU is to reduce network traffic; this demonstrates that the algorithm achieves the goal as expected. Interestingly, Belady underperforms the baseline LFU and incurs more network traffic, because Belady keeps prefetching entire segments for future queries, which takes little advantage of computation pushdown.

As θ increases, the performance benefit of WLFU decreases. When θ is small, there is little access skewness, so segments with higher pushdown cost are cached, leading to the higher effectiveness of WLFU. When θ is large, the access skewness overwhelms the difference of pushdown cost among segments. In this case, almost all the hot segments are cached, leading to reduced performance gains for WLFU.

7.2.4 *Impact of the Network Bandwidth.*

A major bottleneck in the storage-disaggregation architecture is the network bandwidth between the computation and storage layers, thus in this experiment we study its impact on the performance. We evaluate on the c5n.9xlarge instance and vary the network bandwidth by artificially setting *readRateLimiter* of the S3 client.

The Hybrid Architecture. We first study the effect of the network bandwidth on different architectures. *Pullup* always has the worst

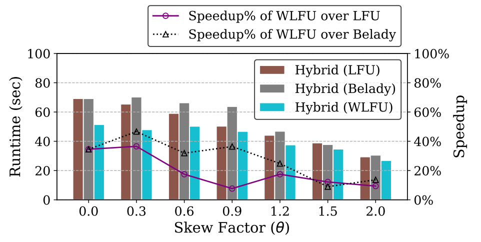
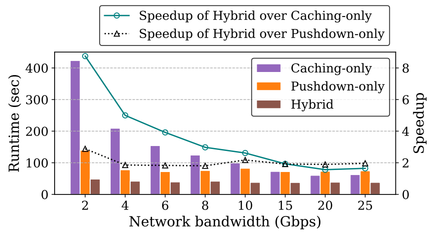
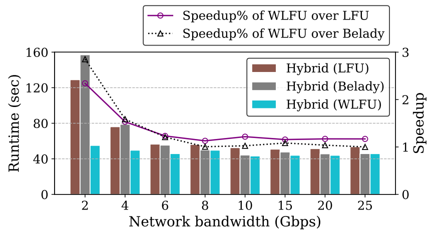


Figure 11: Weighted-LFU Cache Replacement Policy — Runtime comparison of Weighted-LFU (WLFU) and baseline replacement policies (i.e., LFU and Belady) with varying access skewness.



(a) Architectures



(b) Caching Policies

Figure 12: Network Bandwidth — The runtime of different (a) architectures and (b) cache replacement policies as the network bandwidth varies.

performance so we do not include it in this experiment. We use the same workload as Section 7.2.1 with the default cache size and skewness, and report the results in Figure 12(a). *Hybrid* suffers the least from the limitation of the network bandwidth, since it incurs the least network traffic. With throttled network, the more network traffic involved, the higher latency during network transfer, so the performance benefit of *Hybrid* keeps increasing. With 2 Gbps network bandwidth, *Hybrid* achieves 8.7× speedup over *Caching-only*, and 2.9× speedup over *Pushdown-only*.

Weighted-LFU Caching Policy. Next we investigate the impact of the network bandwidth on different caching policies. We use the same workload as Section 7.2.3 with the default cache size and small θ (i.e., $\theta = 0$). As described in Section 7.2.3, with a large θ, segments with a higher access frequency are cached, so both LFU and WLFU cache hot data, leading to similar performance; i.e. WLFU outperforms LFU more with a small θ. In this experiment we set θ to 0. We compare Weighted-LFU, LFU, and Belady caching policies in the *Hybrid* architecture, and report the result in Figure 12(b). Among different caching policies, WLFU suffers the least from the limitation of the network bandwidth. With the network gradually being throttled, the performance benefit of WLFU over LFU becomes more significant, due to the amplification of the benefit from the reduction of network traffic. When the network bandwidth is limited to 2 Gbps, WLFU outperforms baseline LFU and Belady by 2.3× and 2.9×, respectively.

7.3 Resource Usage and Cost

In this subsection, we investigate the resource usage and cost of different architectures. We note that a specific pricing model (e.g., AWS S3 Select) may involve various non-technical factors, and thus may not be the best metric in measuring the efficiency of a system. Therefore, we build a *resource usage model* purely based on the usage of CPU and network resources, then we explain how the resource usage model is related to various pricing models.

We consider two kinds of resources: CPU and network. Table 2 compares the network usage of different architectures; these numbers are directly measured at the compute node. *Hybrid* reduces the network traffic by 79% over *Pushdown-only* due to caching and data reuse, and 93% over *Caching-only* because *Hybrid* can push down filtering to the storage layer.

In a cloud database, the compute servers can be either dedicated to a user or shared across multiple tenants (e.g., a serverless model). We estimate the CPU usage in both cases. For the compute server, we monitor the CPU usage using *vmstat*. Since we cannot directly measure the CPU usage for S3 storage servers, we estimate it by running the same pushdown task in the compute node and measuring the CPU usage. With a dedicated server, users pay for the entire server for the duration of query execution; idle but reserved CPUs are considered "used" during query execution. Table 3 shows the CPU usage. *Pullup* and *Caching-only* have no CPU usage of storage since there is no pushdown. *Hybrid* incurs the least CPU usage due to its low runtime.

In the multi-tenant case, users pay only for the CPU-time actually consumed by the query. The CPU usage is shown in Table 4. *Hybrid* reduces the CPU usage by 60% compared to *Pushdown-only*. However, *Hybrid* increases the total CPU usage by 20% over *Caching-only* because of the redundant work performed at both the compute and storage sides during pushdown, including evaluating the same filter predicate (cf. Section 4.3) and data parsing. The extra CPU resource consumption of pushdown is relatively small.

As a conclusion, *Hybrid* achieves higher performance using comparable CPU resource and much less network resource.

How the resource usage model translates to monetary cost depends on the use case of the system. We identify the following three common use cases:

(1) Users deploy the cloud storage service on their own infrastructures. Examples include Ceph [2] and MinIO [5]. Private clouds fall into this category.

Table 2: Network Usage (GB) of different architectures.

Architecture	Pullup	PD-only	CA-only	Hybrid
Usage	460.9	37.1	112.6	7.9

Table 3: CPU Usage (with dedicated compute servers) — CPU time (in minutes) of different architectures (normalized to the time of 1 vCPU).

Architecture	Pullup	PD-only	CA-only	Hybrid
Compute	249.6	48.5	70.3	23.2
Storage	0.0	31.1	0.0	7.4
Total	249.6	79.6	70.3	30.6

Table 4: CPU Usage (with multi-tenant compute servers) — CPU time (in minutes) of different architectures (normalized to the time of 1 vCPU).

Architecture	Pullup	PD-only	CA-only	Hybrid
Compute	41.8	15.9	15.7	11.5
Storage	0.0	31.1	0.0	7.4
Total	41.8	47.0	15.7	18.9

(2) Users adopt public cloud storage like S3, computation occurs in **a different data center** from the storage.
(3) Users adopt public cloud storage like S3, computation occurs in **the same data center** as the storage.

In case 1, users are already paying for the storage infrastructure. Pushdown is simply exploiting otherwise underutilized resources within the storage layer and does not introduce extra cost.

In case 2, most cloud service providers charge significantly more when the data is transferred outside of a region. For example, AWS charges $0.09/GB for data transferred outside the cloud and $0.02/GB for data transferred across data centers. In this case, the traffic reduction of *Hybrid* leads to significant cost savings.

In case 3, the cost depends on the pricing model of computation pushdown. We observe that S3 Select charges significantly more for storage-layer computation than regular EC2 computation. As a result, *Hybrid* is more expensive than *Caching-only* in S3 Select today. We believe future systems will offer a more fair pricing model, making pushdown computation more attractive in terms of cost.

8 RELATED WORK

In this section, we discuss further related work that has not been discussed earlier in the paper.

Cloud Databases. Modern cloud databases adopt an architecture with storage disaggregation. This includes conventional data warehousing systems adapting to the cloud (e.g., Vertica [36] Eon [45] mode) as well as databases natively developed for the cloud (e.g., Snowflake [21, 48], Redshift [30], Redshift Spectrum [8], Athena [7]).

Besides data warehousing systems, transactional databases also benefit from storage-disaggregation. AWS Aurora [46, 47] is an OLTP database deployed on a custom-designed cloud storage layer where functionalities including log replay and garbage collection are offloaded to the storage layer. The disaggregation of computation and storage also allows each component to easily adapt the workload requirements dynamically.

Computation Pushdown. The technique of computation pushdown has been explored in multiple research areas, both within and beyond database systems, including in-cloud databases, database machines, Smart Disks/SSD, and processing-in-memory (PIM).

Many in-cloud databases push certain computation tasks near the data source, such as AWS Redshift Spectrum [8], S3 Select [31], and PushdownDB [53]; these systems use software techniques to implement pushdown functionalities. Recently, AWS Advanced Query Accelerator (AQUA) [12] announced to use special hardware accelerators (i.e., AWS Nitro chips [6]) to implement pushdown functions with faster speed and lower energy consumption.

Database machines have emerged since the 1970s. Many of them push computation to storage via special hardwares. The Intelligent Database Machines (IDM) [44] pushes most DBMS functionalities to the backend machine which sits closer to disks. In Grace [25], multiple filter processors are connected to disks to perform selection and projection. IBM Netezza data warehouse appliances [23] enable pushdown of selection, projection, and compression to disks through FPGA-enabled near-storage processors. Finally, Exadata Cells in the storage layer of Oracle Exadata Database Machine [49] support operations including selection, projection, bloom join, and indexing. To the best of our knowledge, none of these systems support fine-grained hybrid caching and pushdown like *FPDB* does.

Smart Disks/SSD is another line of research exploiting computation pushdown. Projects including Active Disks [39] and IDISKS [33] have investigated the idea of pushing computation to magnetic storage devices. Summarizer [35] and Biscuit [29] are near-data processing architectures supporting selection pushdown to the SSD processor. Both filtering and aggregation are pushed to smart SSDs [22] and near-storage FPGAs [26, 51] to accelerate relational analytical processing. AQUOMAN [52] supports pushing down most SQL operators including multi-way joins to SSDs. Again, none of these systems support fine-grained hybrid caching/pushdown like *FPDB* does.

Moreover, some recent work explored *processing-in-memory* (PIM) to push computation into DRAM or NVM. Modern 3D-stacked DRAM implemented a logic layer underneath DRAM cell arrays within the same chip [20], avoiding unnecessary data movement between memory and the CPU [27]. Kepe et al. [34] presented an experimental study focusing on selection in PIM.

Client-server Architectures. Many distributed databases adopt a client-server architecture. These systems can move the data to the client where the query is initiated (i.e., caching); alternatively, the query can be moved to the servers where the data resides (i.e., pushdown). Plenty of research focused on query optimization by combining both. Franklin et al. [24] introduced Hybrid Shipping which executes some query operators at the client-side, where the query is invoked, and some at the server-side, where data is stored. Garlic [41] extended this approach by pushing query operators to heterogeneous data stores. MOCHA [40] supports pushing both query operators and their code to remote sites where these operators are not even implemented. The key difference between *FPDB* and these systems is that *FPDB* is built for a storage-disaggregation architecture in a cloud database, where the computation power inside the cloud storage is more limited than the servers in a client-server architecture.

9 CONCLUSION

This paper presented *FPDB*, a cloud-native OLAP database that combines the benefits of caching and computation pushdown in a storage-disaggregation architecture through fine-grained hybrid execution. We explored the design space in a hybrid query executor and designed a novel *Weighted-LFU* cache replacement policy specifically optimized for the disaggregation architecture. Evaluation on the Star Schema Benchmark demonstrated that the hybrid execution can outperform both *Caching-only* and *Pushdown-only* architectures by 2.2×.

REFERENCES

[1] 2012. Akka. https://akka.io/.
[2] 2012. Ceph. https://ceph.io/.
[3] 2016. Apache Arrow. https://arrow.apache.org/.
[4] 2016. Apache Parquet. https://parquet.apache.org/.
[5] 2016. MinIO. https://min.io/.
[6] 2017. AWS Nitro System. https://aws.amazon.com/ec2/nitro/.
[7] 2018. Amazon Athena — Serverless Interactive Query Service. https://aws.amazon.com/athena/.
[8] 2018. Amazon Redshift. https://aws.amazon.com/redshift/.
[9] 2018. Amazon S3. https://aws.amazon.com/s3/.
[10] 2018. Gandiva: an LLVM-based Arrow expression compiler. https://arrow.apache.org/blog/2018/12/05/gandiva-donation/.
[11] 2018. Presto. https://prestodb.io/.
[12] 2020. AQUA (Advanced Query Accelerator) for Amazon Redshift. https://pages.awscloud.com/AQUA_Preview.html/.
[13] 2020. Azure Data Lake Storage query acceleration. https://docs.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-query-acceleration/.
[14] 2020. Presto documentation, Alluxio Cache Service. https://prestodb.io/docs/current/cache/alluxio.html/.
[15] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press.
[16] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. 1383–1394.
[17] Joe Armstrong. 1996. Erlang—a Survey of the Language and its Industrial Applications. In *Proc. INAP*, Vol. 96.
[18] L. A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM System Journal* 5, 2 (1966), 78–101.
[19] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2016. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures* 45, C (2016).
[20] Hybrid Memory Cube Consortium. 2014. HMCSpecification2.1.
[21] 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 *SIGMOD*. 215–226.
[22] 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 *SIGMOD*. 1221–1230.
[23] Phil Francisco. 2011. The Netezza Data Appliance Architecture.
[24] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. 1996. Performance Tradeoffs for Client-Server Query Processing. *SIGMOD Record* 25, 2 (1996), 149–160.
[25] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. 1986. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *VLDB*. 209–219.
[26] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing. In *HPCA*. 126–137.
[27] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. 2018. Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. *arXiv preprint arXiv:1802.00320* (2018).
[28] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD Record* 23, 2 (1994), 243–252.
[29] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *ISCA*. 153–165.
[30] 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 *SIGMOD*. 1917–1923.
[31] Randall Hunt. 2018. S3 Select and Glacier Select – Retrieving Subsets of Objects. https://aws.amazon.com/blogs/aws/s3-glacier-select/.
[32] Sang-Woo Jun, Shuotao Xu, and Arvind. 2017. Terabyte Sort on FPGA-accelerated Flash Storage. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 17–24.
[33] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. 1998. A Case for Intelligent Disks (IDISKs). *SIGMOD Record* 27, 3 (1998), 42–52.
[34] Tiago R. Kepe, Eduardo C. de Almeida, and Marco A. Z. Alves. 2019. Database Processing-in-Memory: An Experimental Study. *VLDB* 13, 3 (2019), 334–347.
[35] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing Near Storage. In *MICRO*. 219–231.
[36] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *VLDB* 5, 12 (2012), 1790–1801.
[37] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *VLDB* 3, 1-2 (2010), 330–339.
[38] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. 237–252.
[39] Erik Riedel, Christos Faloutsos, Garth A Gibson, and David Nagle. 2001. Active disks for large-scale data processing. *Computer* 34, 6 (2001), 68–74.
[40] Manuel Rodríguez-Martínez and Nick Roussopoulos. 2000. MOCHA: A Self-Extensible Database Middleware System for Distributed Data Sources. In *SIGMOD*. 213–224.
[41] Mary Tork Roth and Peter M. Schwarz. 1997. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB*. 266–275.
[42] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. *VLDB* 12, 12 (2019), 2170–2182.
[43] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive — A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE*. 996–1005.
[44] Michael Ubell. 1985. The Intelligent Database Machine (IDM). In *Query processing in database systems*. 237–247.
[45] Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. 2018. Eon Mode: Bringing the Vertica Columnar Database to the Cloud. In *SIGMOD*. 797–809.
[46] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. 1041–1052.
[47] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, et al. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *SIGMOD*. 789–796.
[48] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an Elastic Query Engine on Disaggregated Storage. In *NSDI*. 449–462.
[49] Ronald Weiss. 2012. A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server. *Oracle White Paper.* (2012).
[50] Matthew Woicik. 2021. Determining the Optimal Amount of Computation Pushdown for a Cloud Database to Minimize Runtime. *MIT Master Thesis* (2021).
[51] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: an Intelligent Storage Engine with Support for Advanced SQL Offloading. *VLDB* 7, 11 (2014), 963–974.
[52] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind Arvind. 2020. AQUOMAN: An Analytic-Query Offloading Machine. In *MICRO*. 386–399.
[53] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 Computation. In *ICDE*. 1802–1805.