Noname manuscript No.
(will be inserted by the editor)

FlexPushdownDB: Rethinking Computation Pushdown for Cloud OLAP DBMSs

Yifei Yang · Xiangyao Yu · Marco Serafini · Ashraf Aboulnaga · Michael Stonebraker

Received: date / Accepted: date

Abstract Modern cloud-native OLAP 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. Computation pushdown is a promising solution to tackle this issue, which offloads some computation tasks to the storage layer to reduce network traffic. This paper presents *FlexPushdownDB*(*FPDB*), where we revisit the design of computation pushdown in a storage-disaggregation architecture, and then introduce several optimizations to further accelerate query processing.

First, FPDB supports *fine-grained hybrid query execution* to combine the benefits of caching and computation pushdown. Within the cache, FPDB introduces a novel *Weighted-LFU* cache replacement policy that takes into account the cost of pushdown computation. Second, we design *adaptive pushdown* as a new mechanism to avoid throttling the storage-layer computation during pushdown, which pushes the request back to the computation layer at runtime if the storage-layer computational resource is insufficient. Finally, we derive a general principle to identify pushdown-amenable computational tasks, by summarizing common patterns of pushdown capabilities in existing systems, and further propose two new pushdown operators, namely, *selection bitmap* and *distributed data shuffle*. Evaluation on SSB and TPC-H shows each optimization can improve the performance by 2.2×, 1.9×, and 3× respectively.

Keywords OLAP · Cloud databases · Caching · Computation pushdown · Adaptive query processing · Query optimization

Yifei Yang
University of Wisconsin Madison, Madison, USA
E-mail: yyang673@wisc.edu

Xiangyao Yu
University of Wisconsin Madison, Madison, USA
E-mail: yxy@cs.wisc.edu

Marco Serafini
University of Massachusetts-Amherst, Amherst, USA
E-mail: marco@cs.umass.edu

Ashraf Aboulnaga
University of Texas at Arlington, Arlington, USA
E-mail: ashraf.aboulnaga@uta.edu

Michael Stonebraker
Massachusetts Institute of Technology, Cambridge, USA
E-mail: stonebraker@csail.mit.edu

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 [72]. *Computation pushdown* is a promising solution to mitigate the network bottleneck, where some computation logic is sent and evaluated close to the storage (e.g., selection, projection), resulting in less data returned to the compute layer. Examples include Oracle Exadata [80], IBM Netezza [44],

AWS Redshift Spectrum [4], AWS Aqua [11], and PushdownDB [86]. Although the concept of pushdown computation is absorbed into current cloud databases, its implementation remains somewhat restricted. As a result, there are substantial performance enhancements left unexploited. We identify the following three limitations of existing designs:

First, *caching* is a widely adopted technique, which keeps the hot data in the compute layer to reduce the amount of data transferred between the compute and storage layers. Examples include AWS Redshift [51], Snowflake [42, 79], Databricks [23], and Presto with Alluxio cache service [26]. However, existing cloud DBMSs consider caching and computation pushdown as *orthogonal*. Most systems implement only one of them. Some systems, such as Exadata [80], Netezza [44], Redshift Spectrum [4], and Presto [26] 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.

Second, existing systems decides whether to push down computation *during the query optimization phase statically*. For example, Presto [25] enables pushdown for all filter operators to S3 by setting a flag in the configuration file (*"hive.s3select-pushdown.enabled=true"*). However, pushdown in existing systems does not consider the current storage layer's computational capacity and load at the time when the query is executed. If the storage-layer computation resource is scarce (e.g., due to multi-tenancy), computation pushdown may hurt the performance of a particular query. Unfortunately, it is difficult and sometimes impossible to predict the storage-side computational load ahead of query execution, making it challenging to determine how aggressive computation pushdown should occur.

A third limitation is the lack of a *general principle* that determines which operators are amenable for pushdown. Existing systems empirically consider a subset of relational operators such as selection, projection, and aggregation, due to their ease of implementation and effective traffic reduction. We believe a larger set of operators can benefit from pushdown and a principle should exist to decide which operators are considered.

In this paper, we present *FlexPushdownDB* (*FPDB* in short), which addresses the three limitations above. First, we observe that caching and computation pushdown are *not* orthogonal techniques, and that the rigid dichotomy in existing systems leaves potential performance benefits unexploited. FPDB introduces the concept of *separable operators*, which combines 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. We observe that some of the most commonly-used operators are separable, including filtering, projection, and aggregation. Furthermore, separable operators open up new possibilities for cache management. Traditional cache replacement policies assume that each miss requires loading the 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 depends on how amenable the block is to pushdown—misses that can be accelerated with pushdown (e.g., high-selectivity filters) have a lower cost. We develop a novel cache replacement policy called *Weighted-LFU*, which incorporates caching and pushdown into a unified cost model to predict the best cache admission and eviction decisions. Evaluation shows that the hybrid execution outperforms both caching-only and pushdown-only architectures by 2.2× on the Star Schema Benchmark (SSB) [64]. Weighted-LFU can further accelerate query execution by 37% over the baseline LFU.

Second, we propose an *adaptive query processing* approach that adapts the query plan during execution to consider the current load on storage nodes. We first explore the design space and analyze the theoretical bound — what is the optimal division of the tasks between pushdown and non-pushdown to achieve the best overall performance. Then we design a new mechanism, *adaptive pushdown*, to avoid throttling the storage-layer computation during pushdown. Instead of having the database engine make pushdown decisions, adaptive pushdown lets the storage layer to decide whether to execute an incoming pushdown request, or to *push the request back* to the compute layer. When a pushback happens, the compute layer reads the raw data from the storage layer and processes the task locally. Intuitively, pushdown requests should be executed at the storage layer when sufficient computation resource exists, and pushed back when the storage computation is saturated. We will demonstrate that the proposed mechanism can perform close to the theoretical bound. Evaluation shows that adaptive pushdown outperforms traditional baselines of no pushdown and eager pushdown by both 1.9× on TPC-H [33] benchmark.

Finally, we derive a general principle to identify pushdown-amenable computational tasks, by summarizing common patterns of pushdown capabilities in existing systems. First, pushdown tasks should be *local*—pushdown computation should access data only within a single storage node and not incur data transfer within the storage layer. Second, pushdown tasks should be *bounded*—a pushdown task should require at most linear CPU and memory resources with respect to the accessed data size. This principle preserves the benefits of

storage-disaggregation, and simplifies resource isolation and security in a multi-tenant environment. Following the principle above, we further identify two operators that can benefit from pushdown to the storage layer — *selection bitmap* and *distributed data shuffle*. Evaluation results show that the two new pushdown operators can further accelerate end-to-end query processing on TPC-H by 3.0× and 1.7× respectively.

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, and a novel *Weighted-LFU* cache replacement policy that is specifically optimized for the disaggregated architecture.
- We develop *adaptive pushdown*, which leverages the computation at storage dynamically with the consideration of the storage-layer resource utilization, through a pushback mechanism used to decide if a pushdown task should be executed in the storage.
- We infer a general principle to determine whether an operator is amenable to pushdown from existing systems, and identify two unexplored operators that can benefit from pushdown to the storage layer — *distributed data shuffle* and *selection bitmap*.
- We present the detailed design and implementation of FPDB, an open-source C++-based cloud-native OLAP DBMS, with a storage-layer prototype that supports the proposed pushdown capabilities. 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 introduces the background and limitations of pushdown in existing cloud OLAP DBMSs. After showing an overview of FPDB in Section 3, Sections 4, 5, and 6 discuss hybrid query execution, adaptive pushdown, and proposed pushdown operators, respectively. Section 7 presents implementation details of FPDB. Section 8 evaluates the performance of proposed optimization techniques. Finally, Section 9 discusses the related work and Section 10 concludes the paper.

This paper extends our prior work in [85]. We propose two new optimizations on computation pushdown for modern cloud OLAP databases. We design adaptive pushdown to consider storage-layer resource utilization status at pushdown runtime (Section 5). We propose two new pushdown operators following a general priciple derived from the behavior of existing system (Section 6). Corresponding experiments are added in Section 8. The system prototype, FPDB, is also significantly extended compared to [85], with full TPC-H support, distributed query execution, and an additional storage-layer subsystem in which we implement pushdown with proposed optimizations (Section 7).

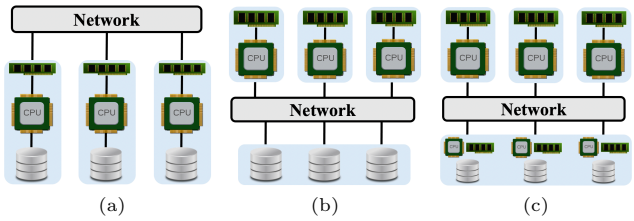


Fig. 1: Distributed Database Architectures—(a) shared-nothing, (b) shared-disk, and (c) storage-disaggregation.

2 Background and Motivation

This section describes the background on the storage-disaggregation architecture and computation pushdown (Sections 2.1–2.2), and the limitations in existing cloud OLAP DBMSs that support pushdown (Section 2.3).

2.1 Storage-disaggregation Architecture

According to the conventional wisdom, *shared-nothing* (Fig. 1(a)) is the most scalable 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 adopt a shared-nothing architecture, many of them choose to disaggregate the compute and storage layers (Fig. 1(c)). This brings benefits of lower cost, simpler fault tolerance, and higher hardware utilization. Many cloud databases have been developed following such an architecture, including Aurora [77, 78], Redshift Spectrum [4], Athena [2], Presto [25], Hive [73], SparkSQL [37], Snowflake [42], and Vertica EON mode [59, 75].

While storage-disaggregation is similar to the conventional *shared-disk* architecture (Fig. 1(b)), they also 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 just like the compute layer. The disaggregation architecture can also provide non-trivial computation in the storage layer, while disks are passive storage devices in the shared-disk architecture.

2.2 Computation Pushdown

The concept of computation pushdown was incubated in database machines since the 1970s. The early systems push computation to storage via special hardware.

Database machines like the Intelligent Database Machine [74], Grace [45], IBM Netezza data warehouse appliances [44], and Oracle Exadata Database Machine [80] move relational operations such as filtering and projection close to disks. Other research areas, including Smart Disks/SSD [43,46,50,58,82,84] and processing-in-memory [47,57] also endorse this spirit.

Cloud databases have emerged in the last decade, with computation and storage disaggregated, especially for analytical queries. The disaggregated architecture supports certain amount of computation within the storage layer, so that some operators can be offloaded to storage to reduce network traffic. The actual computation can happen either on the storage servers (e.g., Aurora [77,78]), or in a different sub-layer close to the storage devices (e.g., S3 Select [29], Redshift Spectrum [4], Presto [25], PushdownDB [86], AQUA [11], Azure Data Lake Storage query acceleration [15]).

2.3 Limitations of Computation Pushdown in Existing Cloud OLAP DBMSs

Even though computation pushdown is widely embraced by existing cloud OLAP DBMSs, there are still restrictions and unexplored opportunities. In the rest of this section, we demonstrate three major limitations to motivate the work in this paper.

2.3.1 Dichotomy between Pushdown and Caching

Caching is a traditional wisdom to speed up query execution in a disaggregated architecture. The system keeps hot data in the local memory or disks of the computation nodes. Cache hits require only local data accesses and are thus much faster than cache misses, which require loading data over the network.

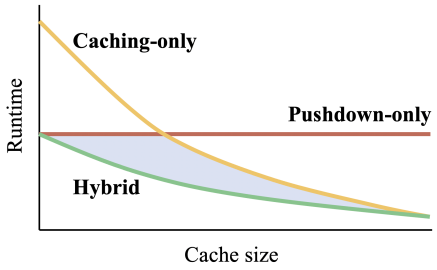


Fig. 2: Performance trade-off between caching, computation pushdown, and an ideal hybrid approach.

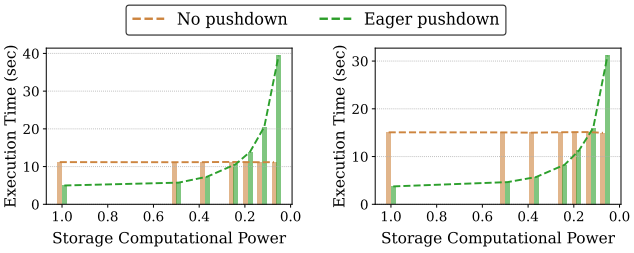
Both pushdown and caching can be leveraged to mitigate the network bottleneck in a disaggregated architecture. Fig. 2 shows the high-level performance trade-off between the two approaches. With only the caching technique, the query execution time decreases as the cache size increases, due to a higher hit ratio. Using only the pushdown technique, 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 push down computation for the rest, which is shown as the bottom line in Fig. 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, considering the two techniques as orthogonal. The storage layer keeps additional "external" tables that can be queried using computation pushdown. No system, to the best of our knowledge, can utilize both caching and pushdown within the processing of a single table in a fine granularity.

2.3.2 Static Pushdown Decisions at Planning Time

Existing cloud OLAP DBMSs make pushdown decisions during the query optimization phase — the query plan is split into two pieces, with the pushdown portion executed in the storage layer, and the rest executed in the compute layer. Once a pushdown decision is made, it cannot be changed at runtime — a pushdown task will always be executed at storage. In other words, pushdown tasks are processed *eagerly*.

Eager pushdown may not always benefit query processing. Since the storage layer is shared by multiple tenants, the amount of available computation resource for each request may vary. To build a deeper understanding, we prototype a S3-like [5] object storage layer within FPDB, and measure the performance of pushdown using standard benchmark queries (TPC-H [33], see Section 8.1 for detailed setups) in different storage-layer resource utilization conditions. Fig. 3 presents the results of two sample queries (Q1 and Q19). No pushdown is included as a baseline for comparison.



(a) Q1 (b) Q19

Fig. 3: Performance of *no pushdown* and *eager pushdown* on Sample Queries — Q1 and Q19 in TPC-H.

For both queries, the performance of no pushdown is independent of the available storage-layer computational power (we regard full storage-layer computational power as all CPU cores at the storage node are available for pushdown tasks, see Section 8.3). Eager pushdown outperforms no pushdown when the storage has abundant computational resource since data transfer is reduced. However, when storage-layer computational resource is insufficient, query execution starts to suffer from the slowdown of pushdown execution. When the storage layer is saturated, eager pushdown underperforms no pushdown, and pushdown execution becomes the major performance bottleneck overall.

An ideal solution should consider the storage-layer computational resource utilization status, and *adaptively* adjust how aggresive the pushdown is. Intuitively, when the storage system is idle on computation, more pushdown tasks should be placed and executed at the storage layer to speed up query processing. Conversely, when the storage system is under heavy load, the DBMS should be more inclined to execute operators at the local compute nodes instead of offloading them to the storage, to avoid throttling storage-layer computation.

2.3.3 Empirically Adopting Pushdown Operators

Existing systems design and implement pushdown features empirically, which end up picking a customized set of pushdown operators respectively. For example, functionalities including selection, projection, and scalar aggregation are supported by almost all existing pushdown systems, grouped aggregation is favored by Redshift Spectrum [4], and pushdown of Bloom filters is introduced in PushdownDB [86].

We aim to conduct a comprehensive analysis of the design space, to identify key characteristics that contribute to the suitability of a pushdown operator. We closely examine the behaviors of existing OLAP DBMSs that offer pushdown support and categorize pushdown operators based on their key features. By deriving a shared pattern from these observations, we can establish a general principle that will potentially facilitate the discovery of new pushdown operators.

3 System Overview

In this section, we demonstrate the high-level system architecture of FPDB (Fig. 4), including both the basic designs and several enhancements.

Hybrid Pushdown with Caching. Fig. 4(a, b) shows the traditional caching-only and pushdown-only designs. In a hybrid architecture (Fig. 4(c)), FPDB stores the hot input data in the local cache (i.e., main memory or disk) to take advantage of fast IO, and keeps the cold input data in the external cloud storage and utlize pushdown computation to reduce network traffic. FPDB contains the following two main modules to enable such hybrid query execution:

The *hybrid 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 computation tasks like filters and aggregations to the storage layer for uncached data. The two portions are then merged and fed to downstream operators. Section 4.1 will discuss the details of how operators are separated and merged.

The *cache manager* determines what data should be cached in the compute 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. Section 4.2 will present detailed cache admission and eviction policies.

Adaptive Pushdown. As discussed in Section 2.3.2, blindly executing all pushdown tasks in the storage layer may throttle the storage-layer computational resources, which may hurt the overall query performance. Ideally, only a portion of tasks that the pushdown engine can sustain are offloaded. For the rest computation tasks, the accessed raw data is returned to the compute nodes and no pushdown occurs. FPDB is enhanced by an *adaptive pushdown arbitrator* deployed in the storage layer (Fig. 4(d)). The adaptive pushdown arbitrator is responsible for determining how aggressive pushdown computation should occur — whether a particular pushdown task should be accepted and executed in the storage, through effective heuristics. Section 5 will discuss the details of the adaptive pushdown mechanism.

Advanced Pushdown Operators. We further enhance FPDB by designing and implementing pushdown operators that are not investigated deeply but can benefit query processing (Fig. 4(e)). Besides the conventional pushdown operators that are supported by existing systems (e.g., selection, projection, aggregation, bloom filter, etc.), FPDB additionally supports offloading *distributed data shuffle* and *selection bitmap* operations to the storage layer. The two proposed pushdown operators are discovered following a general principle which decides whether an operator is amenable to pushdown. We will show the details of the advanced pushdown operators as well as the principle in Section 6.

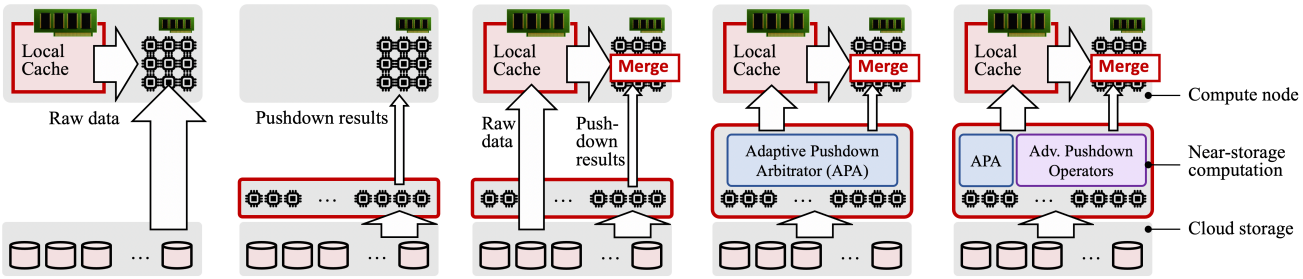


Fig. 4: System Architectures — High level architectures of baseline solutions (caching-only (a), pushdown-only (b)), and three proposed enhancements deployed within FPDB (hybrid execution (c), adaptive pushdown (d), and advanced pushdown operators (e)).

4 Hybrid Pushdown and Caching Execution

In this section, we present the solution of fine-grained hybrid query execution with pushdown and caching, and a new cache replacement policy specifically designed for the storage-disaggregation architecture.

4.1 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 the module and illustrates how the system works through an example query.

4.1.1 Design Choices

Designing the hybrid query executor 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 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 on the raw table data since it is adopted more widely in existing OLAP DBMSs [51,42,79,23,26] — the system usually deploys a data cache in a granularity such as tables, pages, blocks, etc.

Storage and Caching Granularity. FPDB stores tables in object cloud storage service. Tables are horizontally partitioned based on certain attributes (e.g., primary key, sorted field, timestamp, etc.). Each *partition* is stored as an object 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 [7]) can be potentially different from the raw input data (e.g., CSV, Parquet [10], etc.).

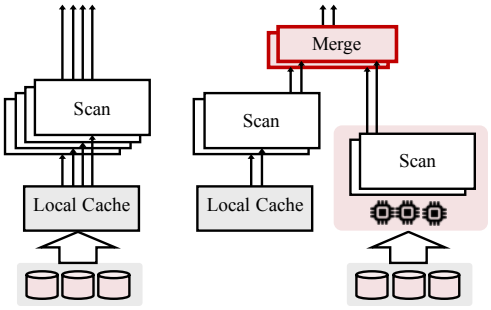
4.1.2 Separable Operators

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



(a) Conventional (b) Separable Query Plan

Fig. 5: Example of a Separable Query Plan — The hybrid query plan contains a parallel *merge* operator that combines the results from cache and computation pushdown.

storage entirely. 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.

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 [86] 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 via pushdown and the segments are then merged in the computation node. Such techniques have been explored in the context of near-storage processing [54] 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.

4.1.3 Separable Query Plan

A query plan is separable if it contains separable operators. Fig. 5 shows an example of a conventional query plan without pushdown (Fig. 5(a)) and the transformed separable query plan (Fig. 5(b)).

A conventional query plan reads all the data from the compute 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 4.1.2.

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.

4.1.4 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 Fig. 6. 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 Fig. 6). Relation R has two attributes A and B, and relation S has two attributes C and D. Four segments are cached locally, as shown in the Local Cache module in Fig. 6.

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 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.

Scan of 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 predi-

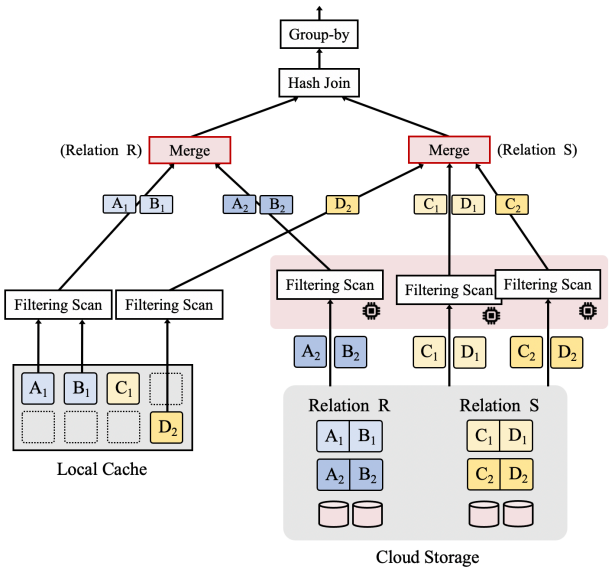


Fig. 6: Separable Query Plan — For the query in Listing 1.

cate. Finally, the local and remote results are combined through a merge operator.

Scan of 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 C_2 and D_2 — 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 4.1.2, 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 without Bloom filter pushdown.

4.1.5 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 have to 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 from the storage layer 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 compute 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 8 will demonstrate the effectiveness of the heuristics with good performance.

4.2 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 noticed a key architectural difference in FPDB that makes conventional cache replacement policies sub-optimal. Conventionally, 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, since 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. We develop a *Weighted-LFU (WLFU)* cache replacement policy based on this observation.

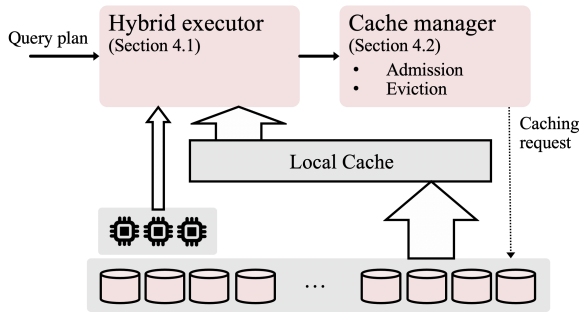


Fig. 7: Integration of Executor and Cache Manager — The cache manager decides what segments should stay in cache.

4.2.1 Integration with Hybrid Executor

Fig. 7 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 during query execution.

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. FPDB adopts the former option to minimize network traffic.

4.2.2 Weighted-LFU Cache Replacement Policy

As discussed previously, 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.

Following this insight, we can tailor the standard LFU specifically for the pushdown context. Instead of incrementing the frequency counter by *1/segment.size* for each access of a segment (assuming the standard LFU is size-sensitive), we increment the counter by a *weight*, which depends on whether the segment can be pushed down and if so, what cost the 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 a row-oriented format (e.g., CSV) — they can be easily accommodated for columnar 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.

5 Adaptive Pushdown

This section presents the detailed design of the adaptive pushdown mechanism, which performs computation pushdown in storage *adaptively*, by taking into account the storage-layer resource utilization status. The high level workflow is depicted in Figure 8, where only a portion of tasks that the pushdown engine can sustain

are offloaded. For the rest computation tasks, only the accessed raw data is returned to the compute nodes and no pushdown occurs. The two portions of computation results are then combined at the compute layer.

One major challenge of adaptive pushdown is that it is hard for the compute nodes to collect accurate statistics from the storage to decide whether a pushable task should be offloaded. Furthermore, even if such information can be accurately collected at planning time, the resource utilization in the storage layer may change at runtime. In this section, we develop a *pushback* mechanism to let the storage layer instead of the compute layer to make the pushdown decision at runtime.

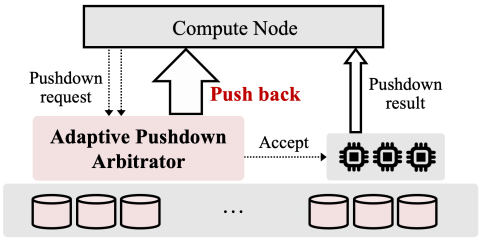


Fig. 8: High Level Workflow of Adaptive Pushdown — The *adaptive pushdown arbitrator* determines whether to accept a pushdown request for execution or push it back.

5.1 Theoretical Analysis

We begin by analyzing the theoretical optimal bound of adaptive pushdown—the optimal division of the computation tasks between pushdown and non-pushdown to achieve the best overall performance. Specifically, we analyze the workload under the following assumptions.

- The workload (which may consist of multiple concurrent queries) contains N pushdown requests submitted to the storage layer in parallel.
- Each pushdown request consumes the same amount of computational resource bw_{pd} when admitted, or the same amount of network resource bw_{pb} when pushed back. The total available CPU and network bandwidth in the storage are BW_{cpu} and BW_{net}.
- The overall execution time of the pushable query plan portion is T_{pd} with pushdown enabled, and T_{npd} with pushdown disabled, where $\frac{T_{npd}}{T_{pd}} = k$. Intuitively, k determines the maximum speedup that any pushdown technique can possibly achieve.

We use the following terms to describe the pushdown decisions made at the storage layer.

- Among N pushdown requests arriving at the storage, n requests are admitted and $N-n$ requests are pushed back.
- The admitted pushdown requests result in an overall execution time of T_{pd_part}, and requests that are pushed back take T_{pb_part} to finish.

Since the admitted pushdown requests at the storage layer and the pushback requests at the compute layer are executed in parallel, the overall execution time of the pushable portion in the query plan can be formulated as follows (Equation 5).

$$T = max\{T_{pd\_part}\,, T_{pb\_part}\} \tag{5}$$

In the optimal case, the storage layer would have a global view of all requests that it will receive ahead of the query execution. Therefore, an optimal split of the pushdown requests for admission and pushback can be constructed. Intuitively, with more pushdown requests admitted at the storage, T_{pd_part} becomes larger, and T_{pb_part} gets smaller, and vice versa. The overall execution time obtains the minimum when these two terms are equal, namely:

$$T_{opt} = T_{pd\_part} = T_{pb\_part} \tag{6}$$

We assume the pushdown tasks are bounded by CPU computation and pushback tasks are bounded by network. Therefore, Equation 6 can be further expanded as follows (Equation 7).

$$\frac{n \cdot bw_{pd}}{BW_{cpu}} = \frac{(N-n) \cdot bw_{pb}}{BW_{net}} \tag{7}$$

T_{pd} and T_{npd} can be expressed similarly (Equation 8).

$$T_{pd} = \frac{N \cdot bw_{pd}}{BW_{cpu}}, \qquad T_{npd} = \frac{N \cdot bw_{pb}}{BW_{net}} \tag{8}$$

Since we know $\frac{T_{npd}}{T_{pd}} = k$ by assumptions, we plug it into Equation 8 and obtain Equation 9.

$$\frac{kN \cdot bw_{pd}}{BW_{cpu}} = \frac{N \cdot bw_{pb}}{BW_{net}} \tag{9}$$

Combining Equation 9 and Equation 7, we can express n as follows (Equation 10).

$$n = \frac{k}{k+1}N \tag{10}$$

Additionally, the optimal execution time can be expressed as follows (Equation 11).

$$T_{opt} = \frac{k}{k+1}T_{pd} = \frac{1}{k+1}T_{npd} \tag{11}$$

Intuitively, a larger k means a higher speedup with pushdown, leading to more tasks executed in the storage (n being larger). Even when the pushdown layer processes tasks slower than the compute layer ($k < 1$),

it can still accelerate the system by offloading some computation. When the pushdown layer is extremely slow or does not exist ($k = 0$), all tasks are pushed back to the compute layer and no pushdown occurs. Note the optimum (Equation 10) can only be approximately satisfied in practice, since the number of pushdown and pushback tasks can only be integers. For example, assume a query submits ten requests to storage, and the optimal division of pushdown and pushback tasks is 7.7 versus 2.3. In practice we have to round them to the closest integers, i.e., execute eight requests at the storage and push back the rest two requests.

5.2 Pushback Mechanism

In our design, the compute nodes always try to offload all pushdown tasks as if the storage has abundant computational resource. When the storage server receives a pushdown request, the *adaptive pushdown arbitrator* (Fig. 8) determines whether the pushdown request should be accepted and executed. If the storage server is busy, the pushdown request is rejected and the computation task is *pushed back*, in which case the raw data is returned and processed at the compute node as if pushdown did not happen. The query plan at the compute layer is then adjusted to accommodate the pushback.

Algorithm 1: Pushback Mechanism of Storage

```
   State: wait queue: Q_wait
          pushdown execution slots: S_exec-pd
          pushback execution slots: S_exec-pb
   Assume: all incoming requests first enter Q_wait
1  while Q_wait is not empty do
2      req = Q_wait.front()
3      t_pd = estimate_pushdown_time(req)
4      t_pb = estimate_pushback_time(req)
5      if t_pd < t_pb then
6          success = try_pushdown(req, S_exec-pd) or
7                    try_pushback(req, S_exec-pb)
8      else
9          success = try_pushback(req, S_exec-pb) or
10                   try_pushdown(req, S_exec-pd)
11     if success then
12         Q_wait.dequeue()
13     else
14         break
```

Algorithm 1 illustrates the pushback mechanism deployed in the storage layer. It is invoked when a new request arrives or a running request completes. The state maintained in the pushdown node includes a *wait queue* (Q_{wait}), which is used to buffer excess pushdown requests when the server is under heavy load, and a finite set of *execution slots* ($S_{exec\text{-}pd}$, $S_{exec\text{-}pb}$) for both pushdown and pushback executions, to help isolate performance among different executions and avoid throttling. We assume all incoming requests first enter Q_{wait}.

For each request in the wait queue (line 1–2), we begin by estimating the execution time for both pushdown and pushback (lines 3–4), which are classified as the faster path and slower path respectively through comparison (line 5). The adaptive pushdown arbitrator first tries to assign the request to the faster path (line 6 and line 9). If the assignment is not successful due to resource contention, e.g., the corresponding execution slots are full, the arbitrator then tries to assign the request to the slower path (line 7 and line 10). If at least one assignment is successful, the request is removed from the wait queue and executed correspondingly, and we start evaluating the next request in the wait queue. The process pauses when both computation and network resources are saturated. The intuition here is that the storage server should balance the resource utilization between CPU and network adaptively.

Time estimation (line 3–4) follows the same methodology adopted by Weighted-LFU cache replacement policy (Section 4.2.2), where the pushdown time (t_{pd}) consists of three components: data scanning, computation, and network transfer of pushdown results. Similarly, the pushback time (t_{pb}) contains data scanning, network transfer of raw data, and compute-layer computation. Note data scanning is included in both pushdown time and pushback time, and will cancel each other when compared in Algorithm 1 (line 5).

We further simplify the time estimation by ignoring compute-layer computation in pushback time, based on two observations. First, in a storage-disaggregated architecture, usually raw data transfer dominates the pushback time so the computation component has little effect (in Section 6.1 we observe most existing pushdown operators are bounded). Second, the storage layer is unaware of the computation bandwidth of the compute nodes, which can vary across different users. This may bring some inaccuracies into the estimation, but overall we observe a near-optimal result on how aggressive pushdown should be and better performance. Section 8.3 will show the pushback mechanism closely aligns with the theoretical analysis in Section 5.1.

5.3 Awareness of Pushdown Amenability

The wait queue deployed in Algorithm 1 manages the arriving pushdown requests in a FIFO order — whether the request will be accepted is purely based on current resource utilization status. However, some requests may benefit more on pushdown compared to other requests (e.g., the request has a selective filter but in-

curs little computation). Intuitively, these pushdown-amenable requests should be given a higher priority to be executed in the pushdown path than the requests that cannot benefit a lot by pushdown.

Consider a scenario where the wait queue contains two requests: r_1 with $t_{pd} = 3$ and $t_{pb} = 4$, followed by r_2 with $t'_{pd} = 1$ and $t'_{pb} = 4$. The two requests have the same estimated pushback time but differs on the estimated pushdown time. Assume at a moment one request can be admitted for pushdown execution and the other needs to be pushed back. Algorithm 1 would first evaluate r_1 and places it into the pushdown path, then evaluate r_2 with it pushed back. However, a better solution would be to push back r_1 instead of r_2, since r_2 incurs a lower execution time by pushdown.

Given a request, we define *Pushdown Amenability* (*PA*) as the potential benefit of pushdown compared to pushback, which can be expressed as Equation 12.

$$PA = t_{pb} - t_{pd} \tag{12}$$

At runtime, the arbitrator keeps the wait queue sorted by the *PA* value of the requests. Pushdown execution always consumes the request with the highest *PA* value, and pushback execution does the reverse. In the example shown above, $PA(r_1) = 1$ and $PA(r_2) = 3$, which means the storage server would consider executing r_2 in the storage and pushes r_1 back. Compared to Algorithm 1, the invariant here is the full utilization of both the computational and network resources. However, the total amount of consumed CPU and network resources are potentially decreased.

6 Advanced Pushdown Operators

In this section, we aim to obtain a comprehensive understanding of the effects of computation pushdown to cloud OLAP DBMSs. We collect the supported pushdown operators in existing cloud DBMSs, and by observing the common patterns, we derive a general principle about whether an operator is amenable to pushdown (Section 6.1). Based on the principle and several observations in query execution of FPDB, we further identify two operators that can benefit from pushdown to the storage (Section 6.2).

6.1 Key Characteristics

Table 1 collects the pushdown support in existing cloud OLAP DBMSs. Selection, projection, and aggregation are mostly considered due to the wide usage and ease of design and development. Beyond this, different systems adopt a customized set of pushdown operators. For instance, grouped aggregation can be pushed to the storage by Redshift Spectrum. PushdownDB supports pushdown of grouped aggregation, top-K, and Bloom filters using existing APIs of S3 Select. However, due to the intrinsic restrictions of S3 Select interfaces, the implementations are not as efficient as they could be. First, pushdown of grouped aggregation has to be processed in two phases, resulting two rounds of data exchange. Second, Bloom filters are required to be serialized explicitly into strings with 0s and 1s, which is space- and computation-inefficient. Finally, PolarDB-X enables pushdown for a specific join type which requires both tables co-partitioned on the join key.

We conclude the following key characteristics from Table 1 that contribute to the suitability of a pushdown operator.

Key Characteristics of Pushdown. *The required storage-layer computation is local and bounded.*

Characteristic 1: Locality. Locality means the computation tasks placed at the storage layer do not incur any network traffic across the storage servers—the traffic occurs only between the storage layer and the compute layer.

Analysis of Operators. Popular pushdown operators like selection, projection, and scalar aggregation comply with locality, since no network traffic is incurred across storage nodes. The same rationale also applies to operators including grouped aggregation, Bloom filter, top-K, sort, where the computation functionality can be performed on each individual data object.

General join does not embrace locality, unless the two joining relations are co-partitioned using the join key. Otherwise the data needs to be shuffled across the network to redistribute the tables. PolarDB-X supports pushdown of co-partitioned joins. Another example of non-local operator is *Merge*, which combines the output of multiple upstream operators (e.g., select, project, aggregate, sort, etc.). Merge requires data exchange within the storage layer since data objects are typically spread across multiple storage severs, making it a non-local operation. As Table 1 shows, none of the existing cloud DBMSs supports pushing merge to the storage.

Potential Advantages. We believe locality is an important characteristic for pushdown operators for three reasons. First, a pushdown environment must support multi-tenancy. Forbidding data exchange across storage servers can reduce the performance variations in pushdown tasks (e.g., due to network interference and queueing). Second, cloud storage must be encrypted using protocols like TLS [69] during data transfer. Local

Table 1: Supported Pushdown Operators in Existing Cloud OLAP DBMSs — Systems supporting pushdown via S3 Select with no additional enhancement are represented as *S3 Select* (e.g., Presto). Marked with *: Pushdown of grouped aggregation and Bloom filters are not efficiently supported by PushdownDB, join pushdown in PolarDB-X requires both tables co-located on the join key, and top-K pushdown in PolarDB-MySQL requires indexes on the sort key.

Operator	Selection	Projection	Scalar Agg.	Grouped Agg.	Bloom Filter	Top-K	Sort	Join	Merge
Redshift Spectrum	✓	✓	✓	✓					
AQUA	✓	✓	✓						
S3 Select	✓	✓	✓						
PushdownDB	✓	✓	✓	✓*	✓*	✓			
Azure Data Lake Query Acceleration	✓	✓	✓						
PolarDB-X	✓	✓	✓	✓		✓	✓	✓*	
PolarDB-MySQL	✓	✓			✓	✓*			

operators avoid the complexity of encryption and decryption across storage servers (e.g., distributing private keys). Third, local pushdown keeps the design of the storage layer simple, since otherwise it needs to support the server-side functionalities of client-server APIs.

Characteristic 2: Boundedness. *Bounded* implies that pushdown tasks should only require at most linear amount of CPU and memory resources with regard to the accessed data size.

Analysis of Operators. Selection, projection, and scalar aggregation are linearly bounded since the CPU consumption is linear to the size of the processed data, and the memory consumption is a constant. Grouped aggregation consumes linear CPU and linear memory capacity. Bloom filter can be regarded as a special regular filter which is thus also linearly bounded. Top-K is typically implemented using a max or min heap which consumes $O(K)$ memory and $O(NlogK)$ execution time, where N is the size of the input data. The variable K is a constant and typically much smaller than N, making the time complexity also linear in N.

The computation complexity of the sort operation is not linearly bounded. The boundedness of the join operator depends on the cardinality of its output. In a key-foreign key join, the output size is bounded by the larger table, which also serves as the upper bound for both memory usage and computational complexity. However, if the join is not an equi-join, it needs be computed using a nested loop, resulting in complexity that grows beyond linear. Only PolarDB-X incorporates join pushdown for co-partitioned tables.

Potential Advantages. Supporting only *bounded* operators preserves the key benefits of storage-disaggregation, where the storage service scales only based on the volume of the stored data, regardless of the computational resource consumption of the workloads. If pushdown operators are not bounded (linearly), the computational resource consumption may grow beyond the scale of the store data, which requires the storage servers to balance between storage and computational needs, which defeats the purpose of storage-disaggregation.

6.2 Proposed Pushdown Operators

Following the key characteristics derived from existing systems, we identify two commonly used operators in modern distributed query processing that can also benefit from pushdown to storage, but have not been deeply investigated previously. First, we observed the hybrid pushdown and caching execution (Section 4) may waste some cached segments when they cannot cover the filter predicates entirely. To tackle this, we leverage late materialization to push down *selection bitmap* instead of the filter predicate to better use the cached data. Second, we noticed that existing pushdown operator are not specifically designed for distributed execution. Therefore, we propose to offload *distributed data shuffle* operation to the storage layer to reduce network traffic.

Selection Bitmap. Late materialization is adopted by columnar OLAP engines broadly, as demonstrated by various previous studies [71, 67, 62, 34, 68]. Selection bitmaps are one common technique embracing late materialization, and DBMSs frequently filter columnar data using selection bitmaps. For instance, when selecting a column based on a filter predicate on another column, the predicate column is read in first to generate a selection bitmap. This bitmap is then used to filter the selection column. Similar ideas appear in joins, where the join columns will generate a bitmap, which is used to select the non-join columns.

Storage-disaggregation opens up new design space and optimization opportunities to better utilize late materialization — bitmap construction and bitmap apply may not happen in the same layer, since modern columnar query engines cache individual columns within the compute nodes [38, 42, 23, 18]. Selection bitmap push-

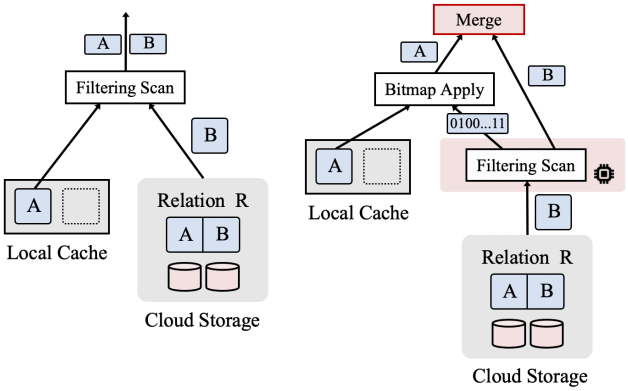
down essentially is a variant of regular filtering pushdown, and hence is both local and bounded. In the following discussion, we will delve deeper into this larger design space by demonstrating with two prevalent cases, and then formulate a comprehensive solution that can be applied more broadly.

Case 1: Selection Bitmap from the Storage Layer. Selection bitmaps can be transferred from the storage to the compute layer when they can only be created at the storage layer — bitmaps are constructed at storage via predicate evaluation, and then leveraged by the compute nodes to filter columns in the local cache.

W.l.o.g., we use the filtering query below to demonstrate how selection bitmap pushdown works, which essentially evaluates a set of filter predicates on attribute set B and returns both columns of both attribute sets A and B. In more general cases, the filtering query can be a subquery in more complicated queries, for example, producing a input joining relation.

```
SELECT A, B FROM R
WHERE [predicates on B]
```

Listing 2: A General Filtering Query (A and B refer two attribute sets).



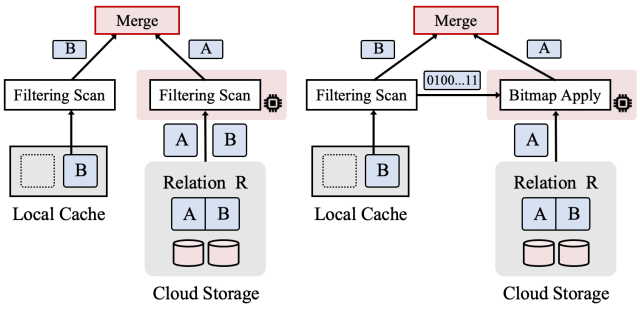
(a) Conventional (b) Selection Bitmap Pushdown

Fig. 9: Selection Bitmap Pushdown (from the Storage Layer) — The selection bitmap constructed at storage can be used to filter cached data at the compute layer.

Assume columns A are stored in the local cache, as Figure 9 shows. Conventionally, the DBMS needs to load the missing columns B to evaluate the filter (Figure 9(a)) (it is also possible that the DBMS pushes down the entire scan and loads both filtered columns A and B). The hybrid solution proposed in [85] also requires the predicate column B loaded to the compute node. Figure 9(b) illustrates the case when selection bitmap pushdown is enabled. The storage layer first constructs a selection bitmap by evaluating the filter predicates on columns B. Then the bitmap is sent to the compute node, where columns A are loaded from the cache and filtered by directly applying the bitmap. At the same time, filtered columns B are returned to the compute node, forming the final result together with filtered columns A.

Case 2: Selection Bitmap from the compute layer. Conversely, when the selection bitmap can be constructed in the compute layer, it can be sent to the storage layer to do filtering without touching the predicate columns from the disks.

Specifically, assume columns B are cached when executing Listing 2, as Figure 10 shows. Conventionally, the storage layer must scan both columns A and B to get the result back to the compute layer (Figure 10(a)). With selection bitmap pushdown, as Figure 10(b) demonstrates, the compute node now can construct a selection bitmap when filtering columns B from the local cache. The bitmap is then sent to storage such that the storage server can perform filtering without loading columns B from disks. Moreover, the CPU cycles used to evaluate filter predicates are eliminated at the storage layer.



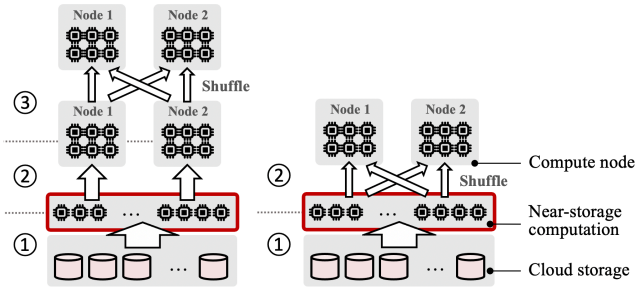
(a) Conventional (b) Selection Bitmap Pushdown

Fig. 10: Selection Bitmap Pushdown (from the compute layer) — Storage can use the compute-layer selection bitmap to perform filtering without touching the predicate column.

Complete Fine-Grained Solution. Selection bitmap pushdown from the storage layer and the compute layer complements each other, as one of them is always applicable depending on what data columns are cached. In practice, filter predicates are often composed of multiple sub-predicates connected by 'AND' or 'OR', and these sub-predicates may themselves be composite. In these situations, a fine-grained execution framework can be used to combine the benefits of both storage-side and compute-side selection bitmaps. For example, on evaluating the predicate "(A or B) and C", assume columns A and B are cached. We can assign the sub-predicate "A or B" to be evaluated at the compute layer, and leave the rest "C" to the storage layer. Selection bitmap are constructed in both layers and then exchanged, al-

lowing for the formation of a complete selection bitmap that corresponds to the input filter predicate, through inexpensive bitwise operations. Subsequently, the complete selection bitmap is used to filter both the cached columns at the compute layer and the uncached columns at the storage layer.

Distributed Data Shuffle. Shuffle is a commonly used operator to redistribute data among compute nodes, for example, when the downstream operator is an equi-join and the two joining relations need to be redistributed based on the join key. In a traditional shared-nothing architecture, shuffle moves data from one node to another based on a partition function (hash-based, range-based, etc), so each data record experiences one network transfer. However, in a storage-disaggregation architecture, as Fig. 11(a) shows, existing systems (e.g., Presto) involve two network transfers—load data from the storage (Step 2) which is pre-filtered within the storage (Step 1, e.g., selection and projection pushdown), and then exchange data across compute nodes (Step 3).



(a) Conventional (b) Shuffle Pushdown

Fig. 11: Distributed Data Shuffle Pushdown — Data is directly redistributed to the target compute node from the storage layer.

Figure 11(b) presents the proposed design where we push the shuffle operators into the storage layer. In this case, the processing of the pushdown tasks is initiated on the storage servers without shuffling (Step 1). Before returning the pushdown results to the compute layer, the data is partitioned and directly forwarded to the appropriate target computation nodes (Step 2). Essentially, the new shuffle design merges steps 2 and 3 in the previous design into a single step, which reduces two network transfers to one.

The compute layer needs to send some key parameters to the storage layer for it to conduct the shuffle operation. These include a partition function, the partition key, and the identifiers of the target compute nodes (such as IP addresses and keys) which the shuffled results are returned to. In our implementation, pushdown requests are sent per data partition, and the parameters used in shuffle processing are attached to each pushdown request when sent to the storage layer. For example, assume there are four compute nodes and eight data partitions in the storage layer. Each compute node would have two corresponding data partitions, and thus send two pushdown requests accordingly. The shuffle operation for a data partition is initiated once its upstream operators are finished (e.g. scan, filter).

There are two approaches for a storage server to transfer shuffled data to target compute nodes: (1) actively pushing the data to the compute nodes, or (2) buffering the shuffled results locally and waiting for compute nodes to request. We chose the latter approach since the target compute nodes may not be immediately ready to receive the data when the storage server issues the transmission. However, the storage server has limited memory space and should not indefinitely write shuffled results to the local buffer. The storage server will set an upper bound on its local buffer size; when the buffer is full, the shuffle process will throttle until the buffer is drained by the target computation node.

Pushdown of distributed data shuffle is *local* since it does not incur network traffic across the storage servers—data is solely transferred from storage to the compute layer. It is also *bounded* because essentially it involves scanning of the input data and assigning records to their corresponding partitions, which consumes CPU and memory resources linearly.

Interact with Cached Data. It is non-trivial to exploit the data in the cache while performing shuffle pushdown, since the shuffle operation changes the data distribution, which means the cached data may not be directly applicable to downstream operators.

The most straightforward method is to ignore the cached data when pushing shuffle to the storage layer — the entire table is redistributed to the compute nodes from the storage. A better solution is to perform the same shuffle function to the cached columns within the compute cluster, and only brings back shuffle results on uncached columns from the storage. The advantages are two-fold. First, in an n-node cluster, a portion of roughly $\frac{1}{n}$ network traffic of redistributed data can be saved, assuming the raw data is initially uniformly distribute into the n nodes and the shuffle function evenly partitions the data across the cluster. This is because in each node, around $\frac{1}{n}$ of the data will be redistributed to the node itself. Second, the network bandwidth within the compute cluster is usually higher than the bandwidth between the computation and storage layer, and reading cached data is more efficient than loading data from the storage devices.

However, it may not be directly applicable to perform the shuffle function over the cached data, which

depends on the existence of the shuffle columns in the cache. To tackle this, we can resort the similar insight of selection bitmap pushdown. When processing the partitioning function, the storage layer can generate a *position vector*, which represents the compute node that each row should be redistributed to. For a n-node compute cluster, each position value requires $log_2 n$ bits, making the position vector a lightweight data structure.

Interact with the Upstream Operator. The rationale aformentioned can be generalized when the shuffle operation is not the direct downstream operator of data scan—it may be performed on intermediate results of computation like filtering and aggregation. The input data produced by the upstream operator can be divided into local portion—produced within the compute layer, and remote portion—produced within the storage layer (i.e. pushdown), which can also abstractly be regarded as cached data and uncached data respectively.

Pushdown Framework. Pushdown within FPDB basically takes the query plan and moves some operators to the storage layer, which is another physical implementation of the same logical plan of no pushdown, such that the result correctness can be guaranteed. In principle, any operator can be pushed down if the storage layer supports it. FPDB pushes down an operator when (part of) the input data can only be accessed in the storage layer to reduce network traffic. Note whether the pushdown actually happens is determined by adaptive pushdown (Section 5). Operators that already exist in the query plan are straightforward to be offloaded—the optimizer adjusts the placement of the pushdown operator to the storage layer (e.g., shuffle). For pushdown operators that are not in the query plan (e.g., selection bitmap), FPDB transforms the query plan to be an equivalent plan using such operators (e.g., a filter operation is transformed as a bitmap-construct operation and a bitmap-apply operation).

7 Implementation

Given that not many open-source cloud-native OLAP DBMSs exist, we decided to implement a new prototype, FPDB, in C++ and make the code publicly available to the community[1]. Fig. 12 shows the distributed query execution framework of FPDB. FPDB adopts a storage-disaggregation architecture. Data is stored persistently in the object storage layer like AWS Simple Storage Service (S3) [5]. FPDB supports querying both row-based (e.g., CSV) and columnar data formats (e.g., Parquet [10]), and widely used benchmarks including SSB, TPC-H, and Join Order Benchmark (JOB) [60].

[1] `https://anonymous.4open.science/r/FlexPushdownDB-Dev-B63D`

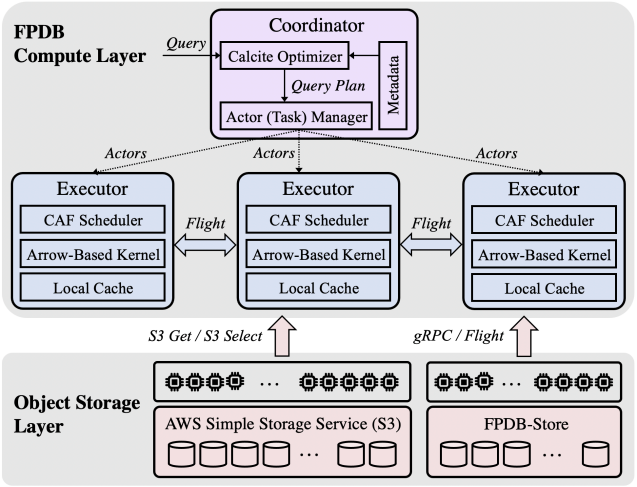


Fig. 12: Distributed Query Execution in FPDB.

7.1 FPDB Compute Layer

FPDB runs on a cluster of AWS EC2 [3] virtual machines, which contains a coordinator node and several executor nodes. When a query arrives, the coordinator is responsible for inspecting the catalog and metadata, parsing the query, and optimizing the query plan. FPDB integrates Apache Calcite [8] as an extendable cost-based query optimizer.

Actor-Based Parallel Processing. FPDB supports distributed parallel query execution using C++ Actor Framework (CAF [41]). CAF is a lightweight and efficient implementation of the Actor model [36], similar to those found in Erlang [39] or Akka [1]. Physical operators are wrapped into actors, which are remotely spawned at the executor nodes by the Actor Manager. A query is composed of a number of actors arranged in a tree. Within each individual executor node, the CAF scheduler multiplexes active actors over all CPU cores on the host machine.

Actors communicate via message passing from producers to consumers — messages flow from leaves to the root. FPDB manages actor behaviors using data messages and control messages. Data messages represent the execution result of the producer, as well as the data input of the consumer, which are sent in a pipelined fashion. Data messages are placed in the shared memory when the producer and consumer actors are in the same node. For data transfer among different nodes, FPDB leverages Apache Arrow Flight [12], which can achieve a much higher usage of the network bandwidth over CAF's builtin cross-node communication. Arrow

Flight enables wire-speed, zero-copy, and serialization-free data transfer by sending data in Arrow IPC format [13], which can be processed directly by FPDB's execution runtime. Control message are used to instruct actors to begin execution, track their completion status, and collect operators' execution metrics. Query execution begins on leaf operators (i.e., scan), and the results are gathered on the *collate* operator on completion.

FPDB's local cache is also implemented as an actor with cache contents managed by the cache replacement policy. The cache actor communicates with other operator actors through message passing to admit and evict data segments. FPDB performs caching using the main memory (RAM) of the executor nodes.

Arrow-Based Runtime. Within the database engine and the cache, we use Apache Arrow [7] 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, *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 extra data conversions.

Arrow provides efficient compute kernels (aggregation, sorting, join, etc.) using optimization techniques including prefetching for cache-efficiency and vectorization via CPU SIMD instructions. Besides, FPDB uses Gandiva [19] for efficient expression evaluation, which is built on top of Arrow and further enhanced by exploiting LLVM and just-in-time expression compilation.

7.2 FPDB Storage Layer

FPDB supports querying data from object storage including AWS S3 and *FPDB-Store*, a S3-like storage prototype with pushdown capabilities discussed in Section 5 and Section 6.

When querying data from S3, FPDB pushes down computational tasks through S3 Select [29], a feature where limited computation can be pushed down onto S3, including projection, filtering scan, and base table aggregation. Regular data retrieval is performed by issuing S3 GetObject [6] requests. We configure rate limits, timeouts, and connections in AWS *ClientConfiguration* high enough to saturate the network bandwidth. Besides, FPDB does not use HTTPS/SSL which incurs extra overhead, as we expect analytics workloads would typically be run in a secure environment.

To enable customized pushdown features, we developed *FPDB-Store*, an open-source storage layer with pushdown support. Data objects are stored on file systems located on locally attached SSDs, which could be accessed by compute nodes through gRPC [20] calls. Arrow Flight is utilized to send pushdown requests to the storage layer and get results back to the compute layer. Each pushdown request contains a serialized query plan instead of a plain SQL dialect [29,15], to avoid redundant query parsing and planning in the storage.

For pushdown operators with bitmaps (e.g., selection bitmap, Bloom filter), FPDB wraps bitmaps into single-column Arrow tables for low serialization overhead. FPDB also avoids two-phase processing [86] when pushing down grouped aggregation to achieve lower latency. In adaptive pushdown, when the storage decides to push back a request, a special Flight error will be returned to the computation layer, where the corresponding compute node will issue a another gRPC call to retrieve the compressed raw data.

8 Evaluation

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

- How does the hybrid pushdown and caching architecture perform compared to baseline pushdown-only and caching-only architectures, and what is the impact of Weighted-LFU on top of it, compared to traditional cache replacement policies?
- How does adaptive pushdown perform under different resource utilization status?
- What is the performance impact of the two proposed pushdown operators?

8.1 Experimental Setup

Hardware Configuration. We conduct all the experiments on AWS EC2 virtual machines. The compute layer is deployed on compute-optimized instances (e.g., c5a.8xlarge) or memory optimized instances (e.g., r5.4xlarge), and FPDB-Store is deployed on storage-optimized instances (e.g., r5d.4xlarge) when used as the storage layer. All instances run the Ubuntu 20.04 operating system.

Benchmark. We use the widely adopted data analytics benchmarks, the Star Schema Benchmark (SSB) [64] and TPC-H [33]. Each table is partitioned into objects of roughly 150 MB when in CSV. We use both CSV and Parquet format in the experiments, and Parquet data is converted from CSV data. When using FPDB-Store

as the storage layer, data objects are evenly distributed to local disks of the storage instances.

Measurement. We measure the execution time and other relevant metrics (e.g., network traffic). For each experiment (i.e., a single query or a batch of sequentially executed queries), we run three times and record the average execution time with the associated metrics.

8.2 Evaluating Hybrid Pushdown and Caching Execution Framework

This section evaluates the performance of the hybrid query execution framework and Weight-LFU cache replacement policy on SSB (SF = 100). 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* [49] 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).

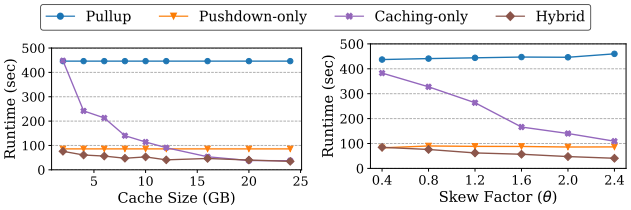
Each experiment sequentially executes a batch of generated queries, with a *warmup phase* and an *execution phase*. The warmup phase contains 50 queries which we found sufficient to warm up the cache. The execution phase contains 50 queries on which we report the performance.

8.2.1 Caching and Pushdown Architectures

We start with comparing the performance of the Hybrid architecture with traditional Pushdown-only and Caching-only architectures. We also add a Pullup baseline, where both pushdown and caching are disabled. We use S3 as the storage layer and leverage S3 Select to perform pushdown computation.

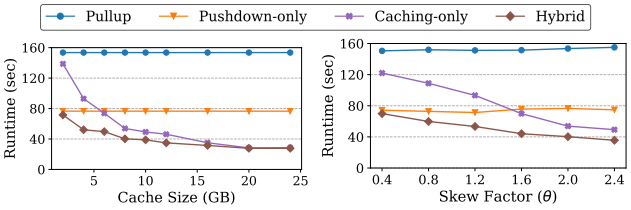
By default, we use LFU cache replacement policy for Caching-only and Hybrid. We report performance varying two parameters: cache size and access skewness of the workload (i.e., θ), on both c5a.8xlarge (32 vCPU, 10 Gbps network) and c5n.9xlarge instances (36 vCPU, 50 Gbps network).

Overall Performance. Fig. 13 shows the runtime comparison on the c5a.8xlarge instance using CSV data. Fig. 13(a) compares different caching/pushdown architectures when the cache size changes. First, we observe that the performance of Pullup and Pushdown-only



(a) Varying Cache Size (b) Varying Skewness

Fig. 13: Performance Comparison (c5a.8xlarge) — The runtime with different (a) cache sizes and (b) access skewness.



(a) Varying Cache Size (b) Varying Skewness

Fig. 14: Performance Comparison (c5n.9xlarge) — The runtime with different (a) cache sizes and (b) access skewness.

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 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 the cache size is bigger than 12 GB, it outperforms Pushdown-only due to the 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 crossing point of Pushdown-only and Caching-only (i.e., roughly 12 GB), Hybrid outperforms both by 2.2×.

Fig. 13(b) shows the performance of different architectures as the access skewness increases (we use a cache of 8 GB which is enough to cache the hot data). 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. Fig. 14(a) compares the performance of different architectures with different cache sizes. The general trends are similar to the case of c5a.8xlarge, and Hybrid consistently outperforms all baselines. With a higher network band-

width, PUSHDOWN-ONLY is 2× faster than PULLUP, which is lower than the speedup on the c5a.8xlarge instance because loading data from the storage 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 Fig. 14(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.

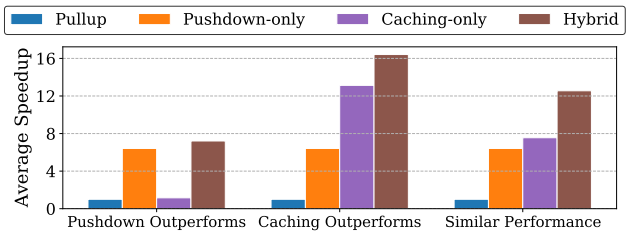


Fig. 15: Per-Query Speedup — The average speedup of each representative case in different architectures.

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 Fig. 15. Although not shown here, the results on c5n.9xlarge have a similar trend.

In all three cases, HYBRID is able to 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 outperforms either baseline significantly. The performance results in Fig. 13 and Fig. 14 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 [26]. For Alluxio, we cache data in main memory, which is consistent with FPDB. We conduct the experiment using the same workload of Fig. 13 with an 8 GB cache and a skewness of 2.0.

Table 2: 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).

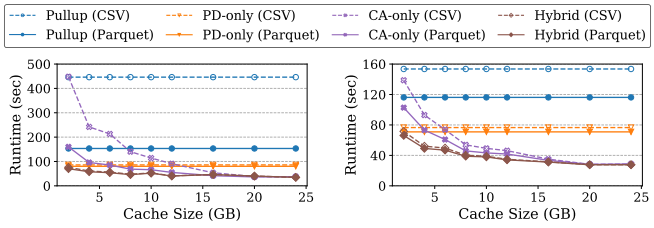
Architecture	Pullup	PD-only	CA-only	Hybrid
Presto	588.7	271.3	536.3	-
FPDB	472.1	111.2	225.7	80.8

The result is shown in Fig. 2. 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 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.

Simulating Parquet Performance. In this experiment, we investigate the performance of FPDB on data in Parquet. 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 [22, 16] or do not support Parquet pushdown at all [15].

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 [81]. 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 [81], 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 Fig. 16. We



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

Fig. 16: Parquet Performance — The runtime estimation of different architectures with different cache sizes on Parquet data. Results on CSV data (Fig. 13(a), 14(a)) are added for reference.

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 Fig. 16(a) and 16(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. 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).

8.2.2 Weighted-LFU Cache Replacement Policy

In this experiment we study the effect of Weighted-LFU (Section 4.2.2) cache replacement policy. Fig. 17 compares it with conventional cache replacement policies including, LFU and Belady [40], an optimal replacement policy that assumes availability of future information. 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 are highly selective (e.g., $lo\_quantity = 10$), while using range predicates on the other attributes (e.g., $lo\_discount < 3$ *or* $lo\_discount > 6$).

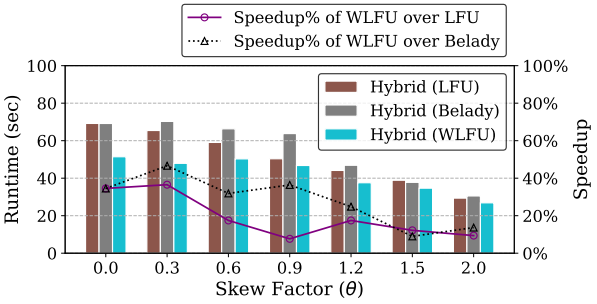


Fig. 17: Weighted-LFU Cache Replacement Policy — Runtime comparison of Weighted-LFU (WLFU) and baseline policies (i.e., LFU and Belady) with varying access skewness.

As Fig. 17 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 and find WLFU can achieve a reduction of 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, both policies cache frequently accessed segments and thus perform similar.

8.3 Evaluating Adaptive Pushdown

In this section, we compare the performance of Adaptive pushdown with traditional No pushdown and Eager pushdown baselines under different storage-layer computational resource conditions, using the TPC-H benchmark (SF = 50). Since S3 does not support execution pushdown tasks adaptively, we use FPDB-Store (Section 7) as the storage layer, which is deployed on r5d.4xlarge instances (16 vCPU, up to 10 Gbps network, and two 300 GB local NVMe SSDs). We emulate the storage-layer computational resource status by varying the number of available CPU cores for pushdown tasks (with a storage computational power of 1 meaning that all CPU cores are available). The compute layer is deployed on r5.4xlarge instances (16 vCPU, up to 10 Gbps network).

Overall Performance. Fig. 18 compares the execution time of Adaptive pushdown with Eager pushdown and No pushdown baselines. When the computational resource at storage is abundant for pushdown execution (i.e., storage computational power is higher than 0.5), Eager pushdown outperforms No pushdown and is only slightly affected by the computational power at storage. As the storage computational power decreases, pushdown execution gets throttled and gradually becomes the major bottleneck, making Eager pushdown underperform No pushdown when the storage-layer computational resource is scarce.

The performance of Adaptive pushdown is consistently better than both baselines. Specifically, when the storage computational power is high, it performs similarly to Eager pushdown, and when the storage server is tiny or under heavy burden, its performance degrades less than Eager pushdown and can still slightly outperform No pushdown. In situations where the storage computational power falls between these two extremes, Adaptive pushdown achieves the best of both worlds. When the performance of No pushdown and Eager pushdown breaks even, Adaptive pushdown outperforms both baselines by 1.5× on average, and a speedup of 1.9× is observed on queries including Q1, Q6, Q8, Q17, and Q19.

With pushdown enabled, the sensitivity on the storage layer's CPU utilization status varies among different queries. Most queries (15 of all) expose a high sensitivity when executed with pushdown. For example, the performance of Eager pushdown of Q1, Q12, Q19, and Q22 is greatly impact by the storage-layer computational power, and starts to degrade even when the storage-layer CPU resource is not scarce. In these queries, the performance improvement of Adaptive pushdown is prominent since the pushable portion of the query plan dominates the end-to-end execution time, and Adaptive pushdown mitigates the issue of resource contention at storage when the pushable subquery plan is processed. For the remaining queries, the performance of Eager pushdown is not very sensitive to the storage-layer computational power (e.g., Q2, Q3, and Q18), where the execution time is dominated by non-pushable operators. In these queries, Adaptive pushdown only shows its superiority when the available computational resource at storage is extremely low. For example, Adaptive pushdown in Q2 outperforms both baselines by 1.2× when the storage-layer computational power is 25%.

Case Study. To get a deeper understanding of the performance benefits, we conduct detailed analysis on two representative queries. We pick Q14 as the query that benefits significantly from computation pushdown, and Q12 as the query that benefits little from pushdown.

We measure the number of admitted pushdown requests at the storage layer in each experiment. Fig. 19 shows the results of the heuristics used in the pushback mechanism (Algorithm 1). For both queries, with the the storage computational power decreasing, fewer pushdown requests are admitted to be executed on the storage server, and more requests are pushed back to the compute layer. Compared to Q12, pushbacks in Q14 are less frequent since it achieves a higher maximal pushdown speedup, such that more tasks are executed in the storage.

We further evaluate the gap between the pushback heuristics and the theoretical optimum (Section 5.1), by comparing the number of actual admitted pushdown requests in the storage with the theoretical result obtained from Equation 10. Overall we observe a very small relative gap between the heuristics and optimal (1% on Q12 and 2% on Q14), and in some cases pushback heuristics achieve the optimal bound (e.g., for Q12 when the storage computational power is less than 25%). This demonstrates that the pushback mechanism is able to find a proper division of the computation tasks between pushdown and non-pushdown.

Fig. 20 compares the incurred network traffic between the storage and compute layers among different pushdown strategies. The network traffic of No pushdown and Eager pushdown both remain consistent, and Eager pushdown reduces network traffic up to an order of magnitude. The network usage of Adaptive pushdown is sensitive to the storage-layer computational power, since it adaptively adjusts the ratio between assigned pushdown and pushback tasks, such that both CPU and network resources at the storage server can be fully utilized.

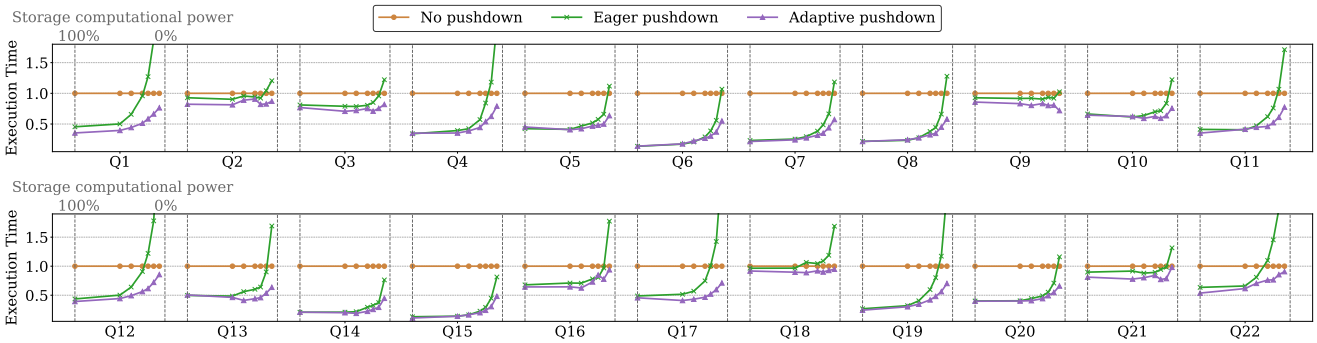
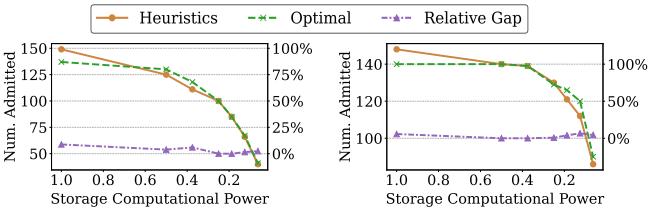
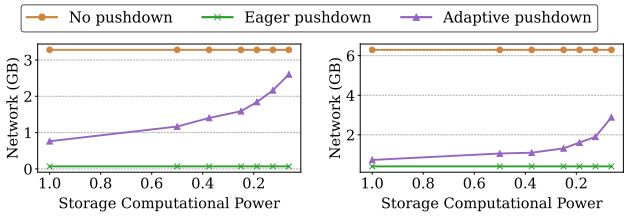


Fig. 18: Performance Evaluation of ADAPTIVE PUSHDOWN on TPC-H (Execution time is normalized to NO PUSHDOWN).



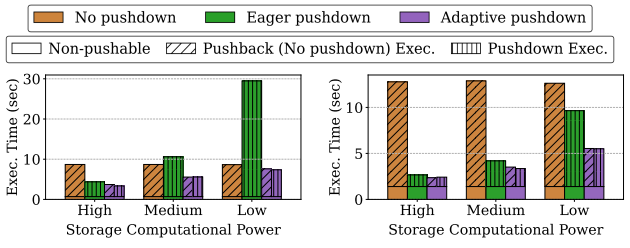
(a) Q12 (Total = 190) (b) Q14 (Total = 156)

Fig. 19: Comparison between Pushback Heuristics and the Theoretical Optimal Bound.



(a) Q12 (b) Q14

Fig. 20: Network Traffic Measured on Two Representative Queries (Q12 and Q14).

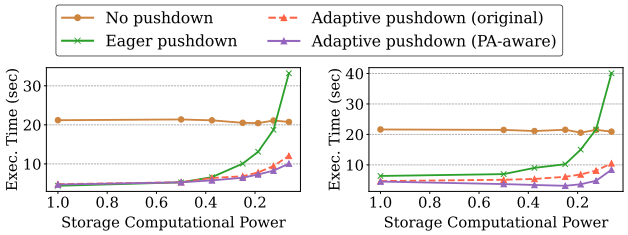


(a) Q12 (b) Q14

Fig. 21: Performance Breakdown on Two Representative Queries (Q12 and Q14).

Fig. 21 shows the performance breakdown on the two representative queries. We present three cases where the storage-layer computational power is high, medium, and low, respectively. The execution time of the non-pushable portion of the query plan remains stable in all cases. Compared to fetching raw data from the storage in NO PUSHDOWN, the overhead of pushback executions in ADAPTIVE PUSHDOWN is consistently smaller since less data is returned. Compared to EAGER PUSHDOWN, ADAPTIVE PUSHDOWN executes pushdown tasks more efficiently since fewer tasks are actually admitted by the storage. We further observe in ADAPTIVE PUSHDOWN that, the performance of pushdown and pushback executions are close, which means our algorithm is able to properly divide pushdown and pushback tasks to balance the usage of CPU and network resources.

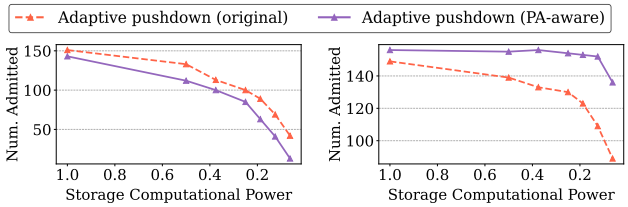
Awareness of Pushdown Amenability. Next we evaluate ADAPTIVE PUSHDOWN when the pushdown requests have different pushdown amenability. Within a single query, pushdown amenability of different requests is similar, since the TPC-H dataset embraces a uniform distribution among different data partitions. Therefore, in this experiment we execute the two representative queries (Q12 and Q14) concurrently to ingest heterogenity of pushdown amenability.



(a) Q12 (b) Q14

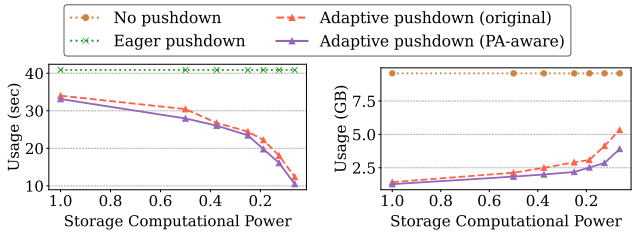
Fig. 22: Evaluation of Awareness of Pushdown-Amenability in Concurrent Executions.

Figure 22 compares the original ADAPTIVE PUSHDOWN and ADAPTIVE PUSHDOWN with awareness of pushdown amenability. We also add NO PUSHDOWN and EAGER PUSHDOWN baselines for references. Both original ADAPTIVE PUSHDOWN and ADAPTIVE PUSHDOWN which is aware of the pushdown amenability (*PA-aware*) outperform the two baselines. Compared to the



(a) Q12 (Total = 190) (b) Q14 (Total = 156)

Fig. 23: Number of Admitted Pushdown Requests at storage in Concurrent Executions.



(a) CPU (b) Network

Fig. 24: Resource Usage of the Storage Layer in Concurrent Executions (CPU usage is measured by the total CPU time that is normalized to the time of 1 vCPU).

original ADAPTIVE PUSHDOWN, PA-aware ADAPTIVE PUSHDOWN further improves the performance of both concurrent queries, where Q12 is accelerated by up to 1.2× and Q14 is improved by up to 1.9×. An interesting observation on Q14 is that the performance is sometimes even improved with lower storage computation power (e.g., 0.3). This is because the performance gap between the two concurrent queries are increased, such that the contention on the non-pushable portion in the compute layer is mitigated — the slower query (Q12) has not entered the non-pushable portion when the faster query (Q14) has completed.

To understand the achieved performance improvement, we trace the number of admitted pushdown requests for both queries respectively, which is shown in Figure 23. Overall, we observe a decrease for the number of admitted pushdown requests on Q12 but an increase on Q14. This is because the requests of Q14 have a potentially larger pushdown benefit, and they are prioritized to be executed at the storage layer. Correspondingly, more requests of Q12 are pushed back to the compute layer. It is interesting to note that the performance of Q12 does not degrade but is even slightly improved, where the reasons are two-fold. First, the difference of the execution time between the pushback path and pushdown path on Q12 is not significant, so a small number of more pushback execution do not hurt the performance. Second, since the requests of Q14 are executed more efficiently, the time spent in the wait queue for the requests of Q12 decreases.

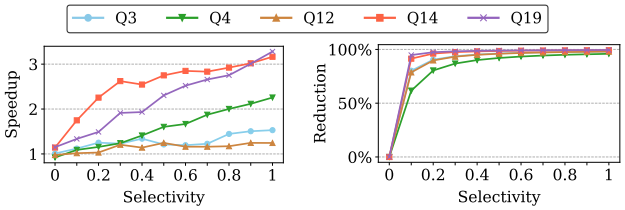
We further investigate the resource usage of CPU and network, which is demonstrated in Figure 24. PA-aware ADAPTIVE PUSHDOWN reduces the CPU usage by up to 15%, and network usage by up to 31%, compared to the original ADAPTIVE PUSHDOWN. The reduction is more significant when storage-layer computational power is lower, since more requests are pushed back to the compute layer, and PA-aware ADAPTIVE PUSHDOWN is able to capture the most beneficial requests that should be pushed back.

8.4 Evaluating Proposed Pushdown Operators

In this section, we evaluate the performance of new pushdown operators, namely, selection bitmap and distributed data shuffle. We implement most existing pushdown operators (selection, projection, aggregation, Bloom filter, etc.) into FPDB-Store to serve as a baseline.

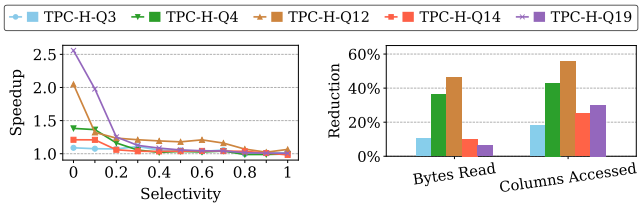
8.4.1 Evaluating Selection Bitmap Pushdown

We first evaluate the performance of selection bitmap pushdown on several representative benchmark queries: Q3, Q4, Q12, Q14, and Q19 (other queries observe similar results), using TPC-H (SF = 50). In each experiment, we vary the selectivity of the filter predicate associated with the fact table `Lineitem`.



(a) Execution Time (b) Network Traffic

Fig. 25: Performance Evaluation of Selection Bitmap Pushdown (Results are normalized to *Pushdown (baseline)*) — The selection bitmap is constructed at the storage layer.



(a) Execution Time (b) Disk Metrics

Fig. 26: Performance Evaluation of Selection Bitmap Pushdown (Results are normalized to *Pushdown (baseline)*) — The selection bitmap is constructed at the compute layer.

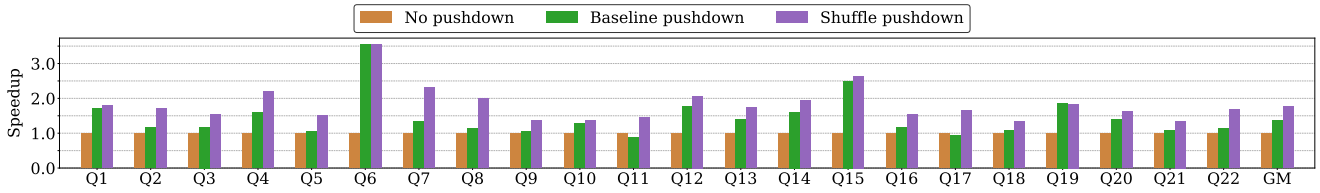


Fig. 27: Performance Evaluation of Distributed Data Shuffle Pushdown on TPC-H (normalized to No pushdown).

Selection Bitmap from the Storage Layer. We first simulate the scenario where the selection bitmap can only be generated in the storage layer. We achieve this by caching only the output columns of the filter operator for the fact table. Predicate columns are not cached. The results are depicted in Figure 25.

As Figure 25(a) illustrates, all queries show an improvement in performance compared to the baseline. Selection bitmap pushdown is most effective in Q14 and Q19. When the filter predicate is non-selective (e.g., 0.9), these two queries perform over 3.0× better than baseline pushdown since transferring selection bitmaps instead of data columns reduces network traffic significantly (over 90% of data transfer is saved, as shown in Figure 25(b)). Q4 also observes a speedup of 2.3×.

When the filter predicate is highly selective, the speedup is less obvious. This is because baseline pushdown returns less data to the compute layer, such that the reduction of data transfer is less significant. Despite this, query execution still gets accelerated. For instance, when the selectivity is 0.1, the speedups of Q14 and Q19 are 1.8× and 1.3× respectively.

The performance gain on Q3 and Q4 is less substantial compared to the other queries. These queries contain more complex operators downstream of pushdown (e.g., more intricate joins and aggregations), leading to a diluted performance benefit.

Selection Bitmap from the Compute Layer. We next emulate the case where the compute-layer selection bitmap can be used to accelerate pushdown execution in the storage layer. In this experiment, only the predicate columns are cached in the compute nodes. Results are displayed in Figure 26.

As demonstrated in Figure 26(a), all queries benefit from selection bitmap pushdown when the filter predicate is selective. For example, when the selectivity approaches 0, pushdown of selection bitmaps outperforms the baseline by 2.0× and 2.6× on Q12 and Q19, respectively. When the filter predicate becomes less selective, the performance gain decreases, since more data is transferred to the compute nodes, which dominates the query execution time.

We also analyze disk metrics by measuring the number of bytes read and the number of columns accessed from the disks, which are illustrated in Figure 26(b). The amount of data scanning is reduced by 36% and 46% on Q4 and Q14 respectively, and by approximately 10% on the rest queries. Additionally, the number of columns accessed of the Parquet data decreases between 18% and 56%. The reduction of data scanning is less substantial compared to column access reduction, because the columns that can be skipped via selection bitmap pushdown are typically highly compressed, such as *l_shipmode*, which only has 7 unique values, and *l_quantity*, of which the value is within a small range between 1 and 50. Conversely, columns that must be transferred are often join keys or have a decimal type, which usually have a low compression ratio.

8.4.2 Evaluating Distributed Data Shuffle Pushdown

Next we evaluate the performance of distributed data shuffle pushdown over TPC-H (SF = 100), as Figure 27 shows. The execution time is normalized to the no pushdown baseline. Across all queries, shuffle pushdown results in an average of 1.3× performance improvement over baseline pushdown, and 1.8× over no pushdown.

Among all 22 queries, we observe the performance improvement on 20 of them, with 15 queries are accelerated by over 1.2×, 10 queries accelerated by over 1.3×, and 6 queries accelerated by over 1.5×. Q7, Q8, and Q17 benefit from shuffle pushdown most significantly, which are improved by more than 1.7×. In these queries, the filter predicates associated with the base tables are not selective, where a major part of the table data still needs to be fetched. Shuffle pushdown is able to eliminate the redistribution of the scanned base table data, which occupies a large portion of the overall execution time. As a result, we observe more than half of the data exchange across the compute nodes is saved compared to baseline pushdown.

Conversely, several queries do not benefit a lot from shuffle pushdown (Q6, Q15, Q19, etc.). These queries typically have highly selective filter predicates on base tables, such that the amount of data transferred from

the storage layer is not significant, and the overhead of data exchange across the compute nodes is negligible.

We further investigate the incurred network traffic in different pushdown configurations. On average shuffle pushdown reduces the consumed network resource by 38%. Specifically, shuffle pushdown reduces the data exchange across the compute layer by 84% on overage, while the network traffic between the compute and storage layers is unaffected. Out of all 22 queries, the incurred data exchange across the compute layer is reduced by over 50% on 20 queries, by over 90% on 16 queries, and by over 99% on 7 queries.

9 Related Work

Cloud Databases. Modern cloud databases adopt an architecture with storage-disaggregation. This includes conventional data warehouses adapting to the cloud (e.g., Vertica [59] Eon [75] mode) as well as databases natively developed for the cloud (e.g., Snowflake [42, 79], Redshift [51], Redshift Spectrum [4], Athena [2]).

Besides OLAP DBMSs, transactional databases also benefit from storage-disaggregation. AWS Aurora [77, 78] 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 concept of computation pushdown has been widely adopted by modern cloud-native databases, such as AWS Redshift Spectrum [4], S3 Select [29], and Azure Data Lake Storage Query Acceleration [15]. Systems such as Presto [25], PushdownDB [86] and FlexPushdownDB [85] support computation offloading via S3 Select [29]. PolarDB-X [24] incorporates sorting and co-located equi-join pushdown. AWS Advanced Query Accelerator (AQUA) [11] uses special hardware accelerators (AWS Nitro chips [14]) to develop pushdown functions.

Beyond cloud databases, computation pushdown has also been investigated in other research fields. Federated databases support querying various data sources which have computation capabilities on their own via connectors like general ODBC or Data Source APIs [31, 27, 28]. In database machines, computation is offloaded to storage through specialized hardware [44, 74, 80, 45]. Smart Disks/SSDs supports executing relational operators within the internal processors [70, 56, 58, 50, 43, 46, 82, 84]. Moreover, the concept of computation pushdown is explored in processing-in-memory (PIM) techniques [47, 57, 61].

Workload Management. General workload management techniques have been intensively studied to make efficient use of system resources in addition to achieving any performance objectives [63, 87, 35], and widely deployed in industrial commercial systems. For example, IBM DB2 workload manager [17] allocates request resources and adjusts the concurrency level based on the source or type of incoming work. SQL Server Resource Governor [32] provides multi-tenancy and resource isolation on single instances of SQL Server that serve multiple client workloads. The same concept can also be found beyond RDBMSs in big-data systems. YARN [76] decouples resource management from the programming model for Hadoop's [9] compute platform. However, existing workload managers like fall short of considering the broader optimization space in an architecture with storage-disaggregation, where a pushdown task can either be executed and pushed back.

Adaptive Query Processing. There is a rich literature on the topic of adaptive query processing [52, 48, 53], which adjusts the query execution dynamically based on more accurate runtime statistics. For example, re-optimization techniques [55, 83] are developed to detect and improve sub-optimal query plans at runtime. [65, 66, 88] designed memory adaptive operators like sorting and hash join. Adaptive query processing is also integrated into modern commercial systems such as Spark [30] and SQL Server [21]. However, optimizing adaptive processing in a pushdown context expose new design constraints including limited computational power in the storage, limited network bandwidth between compute and storage layers, and the coordination between the two layers, which is addressed in Section 5.

10 Conclusion

We presented *FPDB*, a cloud-native OLAP database that optimizes computation pushdown within a storage-disaggregation architecture in several aspects. We propose a hybrid execution mode which combines the benefits of caching and pushdown in a fine granularity. We develop adaptive pushdown that executes pushdown tasks adaptively based on the storage-layer resource utilization. Moreover, we conduct a systematical analysis of existing pushdown operators and propose two new beneficial pushdown operators, selection bitmap and distribute data shuffle. Evaluation on SSB and TPC-H shows each optimizations can improve the performance by 2.2×, 1.9×, and 3× respectively.

References

1. Akka. https://akka.io/
2. Amazon Athena — Serverless interactive query service. https://aws.amazon.com/athena
3. Amazon Elastic Compute Cloud. https://aws.amazon.com/pm/ec2
4. Amazon Redshift Spectrum. https://docs.aws.amazon.com/redshift/latest/dg/c-using-spectrum.html
5. Amazon S3. https://aws.amazon.com/s3
6. Amazon S3 documentation - GetObject. https://docs.aws.amazon.com/AmazonS3/latest/API/API_GetObject.html
7. Apache Arrow. https://arrow.apache.org
8. Apache Calcite. https://calcite.apache.org
9. Apache Hadoop. URL https://hadoop.apache.org
10. Apache Parquet. https://parquet.apache.org
11. AQUA (Advanced Query Accelerator) for Amazon Redshift. https://pages.awscloud.com/AQUA_Preview.html
12. Arrow Flight RPC. https://arrow.apache.org/docs/format/Flight.html
13. Arrow IPC Format. https://arrow.apache.org/docs/format/Columnar.html
14. AWS Nitro System. https://aws.amazon.com/ec2/nitro
15. Azure Data Lake Storage query acceleration. https://docs.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-query-acceleration
16. Ceph. https://ceph.io
17. DB2 Workload Manager. https://www.ibm.com/docs/en/db2/10.1.0?topic=manager-db2-workload
18. Dremio SQL query engine. https://www.dremio.com/platform/sql-query-engine
19. Gandiva: an LLVM-based Arrow expression compiler. https://arrow.apache.org/blog/2018/12/05/gandiva-donation
20. gRPC. https://grpc.io
21. Intelligent query processing in SQL databases. https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing
22. Minio. https://min.io
23. Optimize performance with caching on Databricks. https://docs.databricks.com/en/optimizations/disk-cache.html
24. PolarDB-X. https://www.alibabacloud.com/product/polardb-x
25. Presto. https://prestodb.io
26. Presto documentation - Alluxio cache service. https://prestodb.io/docs/current/cache/alluxio.html
27. Presto documentation - Connectors. https://prestodb.io/docs/current/develop/connectors.html
28. Pushdown computations in PolyBase. https://learn.microsoft.com/en-us/sql/relational-databases/polybase/polybase-pushdown-computation
29. S3 Select and Glacier Select – Retrieving subsets of objects. https://aws.amazon.com/blogs/aws/s3-glacier-select
30. Spark documentation - Adaptive query execution. https://spark.apache.org/docs/latest/sql-performance-tuning.html
31. Spark documentation - Data sources. https://spark.apache.org/docs/latest/sql-data-sources.html
32. SQL Server documentation - Resource Governor. https://learn.microsoft.com/en-us/sql/relational-databases/resource-governor/resource-governor
33. TPC-H Benchmark. http://www.tpc.org/tpch
34. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: How different are they really? In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, p. 967–980 (2008)
35. Aboulnaga, A., Babu, S.: Workload management for big data analytics. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, p. 929–932 (2013)
36. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)
37. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark sql: Relational data processing in spark. In: SIGMOD, p. 1383–1394 (2015)
38. Armenatzoglou, N., Basu, S., Bhanoori, N., Cai, M., Chainani, N., Chinta, K., Govindaraju, V., Green, T.J., Gupta, M., Hillig, S., Hotinger, E., Leshinksy, Y., Liang, J., McCreedy, M., Nagel, F., Pandis, I., Parchas, P., Pathak, R., Polychroniou, O., Rahman, F., Saxena, G., Soundararajan, G., Subramanian, S., Terry, D.: Amazon redshift re-invented. In: Proceedings of the 2022 International Conference on Management of Data, p. 2205–2217 (2022)
39. Armstrong, J.: Erlang—a survey of the language and its industrial applications. In: Proc. INAP, vol. 96 (1996)
40. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM System Journal **5**(2), 78–101 (1966)
41. Charousset, D., Hiesgen, R., Schmidt, T.C.: Revisiting Actor Programming in C++. Computer Languages, Systems & Structures **45**(C), 105–131 (2016)
42. Dageville, B., Cruanes, T., Zukowski, M., Antonov, V., Avanes, A., Bock, J., Claybaugh, J., Engovatov, D., Hentschel, M., Huang, J., Lee, A.W., Motivala, A., Munir, A.Q., Pelley, S., Povinec, P., Rahn, G., Triantafyllis, S., Unterbrunner, P.: The snowflake elastic data warehouse. In: SIGMOD, p. 215–226 (2016)
43. Do, J., Kee, Y.S., Patel, J.M., Park, C., Park, K., DeWitt, D.J.: Query processing on smart ssds: Opportunities and challenges. In: SIGMOD, p. 1221–1230 (2013)
44. Francisco, P.: The Netezza Data Appliance Architecture (2011)
45. Fushimi, S., Kitsuregawa, M., Tanaka, H.: An overview of the system software of a parallel relational database machine grace. In: VLDB, p. 209–219 (1986)
46. Gao, M., Kozyrakis, C.: Hrl: Efficient and flexible reconfigurable logic for near-data processing. In: HPCA, pp. 126–137 (2016)
47. Ghose, S., Hsieh, K., Boroumand, A., Ausavarungnirun, R., Mutlu, O.: Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions. arXiv preprint arXiv:1802.00320 (2018)
48. Gounaris, A., Paton, N.W., Fernandes, A.A.A., Sakellariou, R.: Adaptive query processing: A survey. In: Advances in Databases, pp. 11–25 (2002)
49. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. SIGMOD Record **23**(2), 243–252 (1994)
50. Gu, B., Yoon, A.S., Bae, D.H., Jo, I., Lee, J., Yoon, J., Kang, J.U., Kwon, M., Yoon, C., Cho, S., Jeong, J., Chang, D.: Biscuit: A framework for near-data processing of big data workloads. In: ISCA, p. 153–165 (2016)
51. Gupta, A., Agarwal, D., Tan, D., Kulesza, J., Pathak, R., Stefani, S., Srinivasan, V.: Amazon redshift and the case for simpler data warehouses. In: SIGMOD, p. 1917–1923 (2015)

52. Hellerstein, J.M., Franklin, M., Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, V., Shah, M.A.: Adaptive query processing: Technology in evolution. IEEE Data Engineering Bulletin **23**, 7 – 18 (2000)
53. Ives, Z.G., Halevy, A.Y., Weld, D.S., Florescu, D., Friedman, M.T.: Adaptive query processing for internet applications. IEEE Data(base) Engineering Bulletin **23**, 19–26 (2000)
54. Jun, S.W., Xu, S., Arvind: Terabyte sort on fpga-accelerated flash storage. In: IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 17–24 (2017)
55. Kabra, N., DeWitt, D.J.: Efficient mid-query re-optimization of sub-optimal query execution plans. In: Proceedings of the 1998 ACM SIGMOD international conference on Management of data, pp. 106–117 (1998)
56. Keeton, K., Patterson, D.A., Hellerstein, J.M.: A case for intelligent disks (idisks). SIGMOD Record **27**(3), 42–52 (1998)
57. Kepe, T.R., de Almeida, E.C., Alves, M.A.Z.: Database processing-in-memory: An experimental study. VLDB **13**(3), 334–347 (2019)
58. Koo, G., Matam, K.K., I, T., Narra, H.V.K.G., Li, J., Tseng, H.W., Swanson, S., Annavaram, M.: Summarizer: Trading communication with computing near storage. In: MICRO, p. 219–231 (2017)
59. Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., Bear, C.: The vertica analytic database: C-store 7 years later. VLDB **5**(12), 1790–1801 (2012)
60. Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., Neumann, T.: How good are query optimizers, really? Proc. VLDB Endow. **9**(3), 204–215 (2015)
61. Lim, C., Lee, S., Choi, J., Lee, J., Park, S., Kim, H., Lee, J., Kim, Y.: Design and analysis of a processing-in-dimm join algorithm: A case study with upmem dimms. Proc. ACM Manag. Data **1**(2), 1 – 27 (2023)
62. Lin, Y., Agrawal, D., Chen, C., Ooi, B.C., Wu, S.: Llama: Leveraging columnar storage for scalable join processing in the mapreduce framework. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, p. 961–972 (2011)
63. Niu, B., Martin, P., Powley, W.: Towards autonomic workload management in dbmss. Journal of Database Management (JDM) **20**(3), 1–17 (2009)
64. O'Neil, P., O'Neil, E., Chen, X., Revilak, S.: The star schema benchmark and augmented fact table indexing. In: Technology Conference on Performance Evaluation and Benchmarking, pp. 237–252 (2009)
65. Pang, H.H., Carey, M.J., Livny, M.: Memory-adaptive external sorting. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences (1993)
66. Pang, H.H., Carey, M.J., Livny, M.: Partially preemptible hash joins. In: Proceedings of the 1993 ACM SIGMOD international conference on Management of data, pp. 59–68 (1993)
67. Polychroniou, O., Sen, R., Ross, K.A.: Track join: Distributed joins with minimal network traffic. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, p. 1483–1494 (2014)
68. Raman, V., Attaluri, G., Barber, R., Chainani, N., Kalmuk, D., KulandaiSamy, V., Leenstra, J., Lightstone, S., Liu, S., Lohman, G.M., Malkemus, T., Mueller, R., Pandis, I., Schiefer, B., Sharpe, D., Sidle, R., Storm, A., Zhang, L.: Db2 with blu acceleration: So much more than just a column store. Proc. VLDB Endow. **6**(11), 1080–1091 (2013)
69. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (2018). DOI 10.17487/RFC8446. URL `https://www.rfc-editor.org/info/rfc8446`
70. Riedel, E., Faloutsos, C., Gibson, G.A., Nagle, D.: Active disks for large-scale data processing. Computer **34**(6), 68–74 (2001)
71. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: A column-oriented dbms. In: Proceedings of the 31st International Conference on Very Large Data Bases, p. 553–564 (2005)
72. Tan, J., Ghanem, T., Perron, M., Yu, X., Stonebraker, M., DeWitt, D., Serafini, M., Aboulnaga, A., Kraska, T.: Choosing a cloud dbms: Architectures and tradeoffs. VLDB **12**(12), 2170–2182 (2019)
73. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., Murthy, R.: Hive — a petabyte scale data warehouse using hadoop. In: ICDE, pp. 996–1005 (2010)
74. Ubell, M.: The Intelligent Database Machine (IDM). In: Query processing in database systems, pp. 237–247 (1985)
75. Vandiver, B., Prasad, S., Rana, P., Zik, E., Saeidi, A., Parimal, P., Pantela, S., Dave, J.: Eon mode: Bringing the vertica columnar database to the cloud. In: SIGMOD, p. 797–809 (2018)
76. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E.: Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing, pp. 1–16 (2013)
77. Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., Bao, X.: Amazon aurora: Design considerations for high throughput cloud-native relational databases. In: SIGMOD, p. 1041–1052 (2017)
78. Verbitski, A., Gupta, A., Saha, D., Corey, J., Gupta, K., Brahmadesam, M., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvilli, T., et al.: Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In: SIGMOD, pp. 789–796 (2018)
79. Vuppalapati, M., Miron, J., Agarwal, R., Truong, D., Motivala, A., Cruanes, T.: Building an elastic query engine on disaggregated storage. In: NSDI, pp. 449–462 (2020)
80. Weiss, R.: A technical overview of the oracle exadata database machine and exadata storage server. Oracle White Paper. (2012)
81. Woicik, M.: Determining the Optimal Amount of Computation Pushdown for a Cloud Database to Minimize Runtime. MIT Master Thesis (2021)
82. Woods, L., István, Z., Alonso, G.: Ibex: an intelligent storage engine with support for advanced sql offloading. VLDB **7**(11), 963–974 (2014)
83. Wu, W., Naughton, J.F., Singh, H.: Sampling-based query re-optimization (2016)
84. Xu, S., Bourgeat, T., Huang, T., Kim, H., Lee, S., Arvind, A.: Aquoman: An analytic-query offloading machine. In: MICRO, pp. 386–399 (2020)
85. Yang, Y., Youill, M., Woicik, M., Liu, Y., Yu, X., Serafini, M., Aboulnaga, A., Stonebraker, M.: Flexpushdowndb:

Hybrid pushdown and caching in a cloud dbms. VLDB **14**(11), 2101–2113 (2021)
86. Yu, X., Youill, M., Woicik, M., Ghanem, A., Serafini, M., Aboulnaga, A., Stonebraker, M.: Pushdowndb: Accelerating a dbms using s3 computation. In: ICDE, pp. 1802–1805 (2020)
87. Zhang, M., Martin, P., Powley, W., Chen, J.: Workload management in database management systems: A taxonomy. IEEE Transactions on Knowledge and Data Engineering **30**(7), 1386–1402 (2018)
88. Zhang, W., Larson, P.A.: Dynamic memory adjustment for external mergesort. In: VLDB, vol. 97, pp. 25–29 (1997)