Determining the Optimal Amount of Computation Pushdown to Minimize Runtime for a Cloud Database

by

Matthew Woicik

B.S. Computer Science and Engineering, Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author .....
Department of Electrical Engineering and Computer Science
May 14, 2021

Certified by .....
Michael Stonebraker
Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Determining the Optimal Amount of Computation Pushdown to Minimize Runtime for a Cloud Database

by

Matthew Woicik

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Many cloud databases separate their compute from their storage resources. This design introduces a network bottleneck during query execution that can be mitigated through caching and computation pushdown. Depending on the environmental settings and the specific query, the amount of computation pushdown needed to achieve the optimal runtime may vary. This work presents a runtime prediction model that determines the amount of computation pushdown that results in the fastest runtime and analyzes a real-world implementation of this model on the FlexPushdownDB system running in AWS.

Thesis Supervisor: Michael Stonebraker
Title: Professor

Acknowledgments

I want to thank Mike Stonebraker for making this entire experience possible. It was a pleasure to work with you. I gained a lot of valuable knowledge and matured as a researcher under your guidance.

I also want to thank everyone in the FlexPushdownDB group. Your weekly feedback was invaluable and helped steer the direction of my research. I particularly want to thank Xiangyao Yu for always being available whenever I had research-related questions and Yifei Yang for collaborating on this work.

I also want to thank my girlfriend Addie Chambers for listening to me while I talked through all of my research ideas and offering support throughout the entire process. Without you, I am unsure how I would have gotten through this year, let alone my entire time at MIT. You made this experience and my time at MIT all the more special.

Lastly, I want to thank my family for supporting me throughout my entire academic career. Your support made the entire process so much easier, and I'm very grateful for it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In an increasingly data-rich world, organizations must find ways to efficiently manage information related to their users, products, and expenses in a cost-effective way. Many use database management systems (DBMSs) to accomplish this. DBMSs use queries to update, store, and read data, and offer guarantees for data consistency and fault tolerance.

Before cloud services, organizations would buy the necessary resources for their own DBMS. This incurs a high upfront cost. It is also unclear how much of each DBMS resource is necessary to handle the DBMS's expected load. If the DBMS is under-provisioned, performance can degrade. If it is over-provisioned, the cost of resources is higher than is necessary, and the organization could have handled the same load in a more cost-efficient way.

Cloud services offer the flexibility to purchase compute, memory, and storage resources on-demand so that organizations only pay for what they need. This has led to the use of storage-disaggregation architectures (SDAs) [25], where compute and storage are physically separated and communicate via the network. Organizations can minimize DBMS costs by separating compute from storage in an SDA and scaling them independently to meet a workload's needs, without overprovisioning (and therefore overspending on) the other resources. In a compute-heavy workload, for instance, this separation enables organizations to acquire more computational power without forcing them to purchase more storage space.

1.1 Problem Setting

When using an SDA, the network bandwidth can become a bottleneck because the DBMS may need to transfer large amounts of data between the separated compute and storage resources [15]. A DBMS can use *caching* and *computation pushdown* to alleviate this bottleneck. One DBMS that does this is FlexPushdownDB [25], which will be the focus of Chapter 3.

Caching stores a small amount of data at an intermediate storage layer where it can be accessed quickly, in this case, on the machines with compute. Computation pushdown moves as much of the query execution as possible to where the data is stored, in this case, to the machines with the storage. Even with these techniques, a DBMS still must determine how much computation pushdown will most reduce runtime. The optimal amount of computation pushdown depends on many factors, including the specific query, the data that is present in the cache, the network bandwidth, and the storage's query execution speed.

1.2 Contributions

This thesis makes the following contributions:

1. Optimizes the runtime of FlexPushdownDB, an existing online analytical processing (OLAP) cloud DBMS that uses a storage-disaggregation architecture.

2. Develops a model that uses queries and other environmental settings to predict runtime for a general cloud DBMS using a storage-disaggregation architecture.

3. Validates this runtime model via experimentation on FlexPushdownDB and uses it to determine how much computation pushdown will minimize runtime under varying experimental conditions.

We divide this thesis into three major sections. We first cover important terms and related work in Chapter 2. In Chapter 3, we provide a system overview of FlexPushdownDB, an OLAP cloud DBMS prototype that uses an SDA, along with

novel runtime optimizations introduced by this work. In Chapter 4, we propose and validate a runtime model that identifies how much computation pushdown is optimal for different settings and workloads in a DBMS using an SDA.

Chapter 2

Background

In this chapter, we explain several key concepts and discuss relevant work. First, we will introduce the impact of emerging cloud services on DBMS architectures, including the shortcomings of existing systems. We will then explain existing solutions for these performance bottlenecks, and problems that remain to be solved.

2.1 Cloud Services

Cloud service providers such as Amazon Web Services (AWS) [6] introduce machines that users can rent on an as-needed basis. This pricing model offers higher flexibility to changes in workload and does not have the same upfront costs as on-premise DBMSs. Cloud providers offer many types of machines that can be either general or specialized for tasks such as local computation, memory processing, and data storage. In AWS, these rentable machines are called AWS Elastic Compute Cloud (EC2) [3] instances.

Cloud service providers also offer cheap data storage services. These data storage services typically have performance constraints set by the cloud provider that are not configurable by the external user. When accessing the data in the storage service, a compute machine sends a request to the storage service, and the requested data is sent back to it over the network. AWS's storage service is called AWS Simple Storage Service (S3) [5], and allows users to store, erase, and overwrite files. Using S3 is

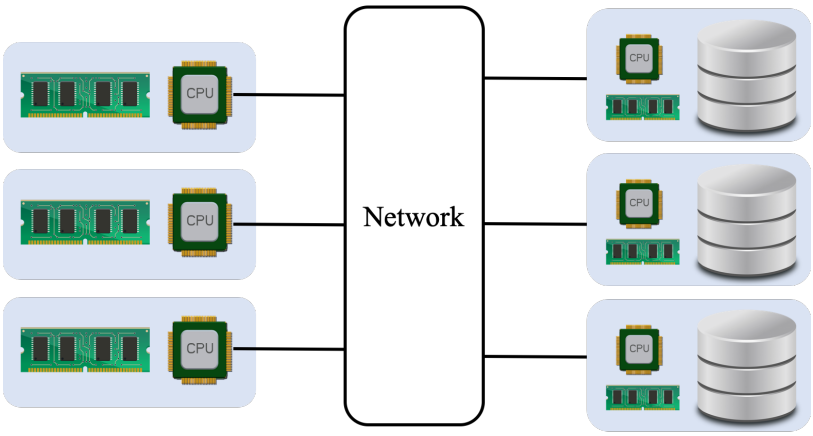


Figure 2-1: An SDA, where the compute layer is the set of machines on the left, and the storage layer is the set of machines on the right. The computational capabilities of the storage layer are more limited than those on the compute layer, as reflected by the size of the different components.

cheaper than renting additional EC2 instances to increase storage capacity.

2.2 Storage-Disaggregation Architecture

Given the difference in pricing between renting machines and storing data in a data storage service, separating these components in a cloud DBMS can lower costs. These savings occur when compute needs do not perfectly scale with storage needs. This has led to a storage-disaggregation architecture (SDA) for cloud DBMSs, where compute and storage are physically separated. The machines with compute handle most of the query execution. The machines with storage handle data storage and can perform a limited amount of DBMS operations before transferring the data across the network. We will refer to these two distinct groups of machines in this architecture as the *compute* and *storage* layers. The compute and storage layers can be seen in Figure 2-1. In a shared disk architecture, compute and storage are also separated. The key difference is that the storage layer in an SDA can perform some types of query processing. This is not the case for a shared-disk architecture.

SDAs are a natural fit for the cloud as they offer the flexibility to easily scale the compute and storage layers. The cloud also provides machines that are suited for either the compute or storage layer tasks. A cloud DBMS can use cloud machines to

make up the compute layer of the SDA and cloud machines or a cloud storage service to make up the storage layer. In the context of AWS, a set of compute-optimized EC2 instances [3] can make up the compute layer, and S3 [5] can act as the storage layer.

2.3 Caching and Computation Pushdown

When separating compute and storage using an SDA, the network can become a bottleneck when sending large amounts of data between the two layers [15]. Two solutions for this problem are caching and computation pushdown.

2.3.1 Caching

Caching stores frequently used data in a fast-access storage layer. The query executor can access data from the cache much more quickly than it can access data from the persistent storage layer. The cache is initially empty and fills up as data is accessed. This process of adding data to the cache is called *warming the cache*. Once the cache is full, some of the cache data may be evicted to add new data. The decisions for which data to add or remove depends on the caching policy used.

A DBMS can use caching to improve data access times for frequently used data, which the system hopes to maintain in the cache. In the context of an SDA, caching stores a subset of the data from the storage layer in the memory of machines in the compute layer.

2.3.2 Computation Pushdown

Computation pushdown in a DBMS moves as much of the query execution as possible closer to where the data is stored. For an SDA, this is done by performing some of the query execution at the storage layer, potentially reducing the amount of data transferred back to the compute layer. This mitigates the bottleneck that an SDA can encounter when operating in an environment with low network bandwidth.

AWS offers computation pushdown to S3 through the S3 Select [18] operation. With S3 Select, users can execute `SELECT`, `FROM`, `WHERE`, and `LIMIT` clauses in S3. This allows database operators such as `filter`, `project`, and `limit`, each of which discards a subset of the original data, to be performed at the storage layer, reducing the amount of data that must be transferred back across the network.

2.4 Existing Work

We will now discuss existing cloud DBMSs that use an SDA. We will then introduce prior work that shows the benefits of computation pushdown in an SDA, and explain how it differs from this work.

2.4.1 Existing Cloud DBMSs

Many cloud-based DBMSs using an SDA employ either caching or computation pushdown to improve their performance. Snowflake [14] and Amazon Aurora [23] are DBMSs that only perform caching. Therefore, they still incur a high data-access time when reading data not in the cache. Snowflake reduces this data transfer time by storing all data in a compressed format.

Amazon Athena [2], Oracle Exadata [24], AWS Redshift Spectrum[4], and PushdownDB [26] only perform computation pushdown. They perform as much computation pushdown as possible in the storage layer when accessing data.

PrestoDB [11] offers the ability either to cache through an external service named Alluxio [1] or to do computation pushdown instead. Alluxio functions as an intermediary layer that takes requests for data from PrestoDB and fulfills them by either returning cached data or accessing data from the storage source that PrestoDB specifies. Alluxio does not currently support computation pushdown, and since Alluxio controls data accesses to the storage layer when it is used, both of these features cannot be integrated simultaneously.

FlexPushdownDB [25] builds upon PushdownDB [26] by incorporating caching in addition to computation pushdown. This makes it unique in that it uses both caching

and computation pushdown simultaneously and tries to use a combination of both techniques to improve its performance.

2.4.2 Optimally using Computation Pushdown

Existing systems discussed in Section 2.4.1 use the same approach for all queries. When a query arrives, these DBMSs operate on the data in the cache (if they are using one) and access the rest of the data from the storage layer (with computation pushdown if they are using it). However, when using caching and computation pushdown, FlexPushdownDB (FPDB) [25] can either perform

1. no computation pushdown,
2. computation pushdown on the data that is **not** in the cache, or
3. computation pushdown on all data.

Using only one approach for all queries may not be optimal for minimizing runtime depending on the

1. cache hit ratio,
2. network bandwidth,
3. storage layer computation pushdown query processing speed,
4. compute layer query processing speed, and
5. computation pushdown's operation selectivity.

Depending on these values, query execution might be fastest using computation pushdown even when all of the data for a query is in the cache. Similarly, a system may be better off not performing computation pushdown during a cache miss.

To the best of our knowledge, no previous work analyzes how much computation pushdown should be performed to minimize runtime while varying all of the dimensions listed above. PushdownDB [26] examines the tradeoff of pushing down computation compared to pulling up the data and performing the operations locally. It does not introduce caching or vary the different processing speeds or network bandwidth. FPDB shows the benefits of performing computation pushdown on data not

present in the cache. However, FPDB does not explore whether or not to use computation pushdown on a per query basis. This is because FPDB has modes of operation that determine whether it uses computation pushdown, and these modes are defined at a system level rather than at a per-query level. FPDB is also only evaluated on queries whose computation pushdown operations have a limited range of selectivity.

This chapter discussed cloud services and how they have led to SDAs for cloud DBMSs. We then examined existing cloud DBMSs and previous work that has explored using computation pushdown in this setting. We will now discuss FPDB [25], a cloud DBMS using an SDA, and the optimizations made by this work to reduce its runtime.

Chapter 3

FlexPushdownDB Optimizations

This chapter will provide a system overview and implementation details of FlexPushdownDB (FPDB) [25], an OLAP cloud DBMS prototype built by Yang et al. using an SDA. We then define the experimental setting for our performance optimizations. Finally, we introduce the novel runtime optimizations contributed to FPDB by this work. The optimizations include improved conversion rate from the data storage format to the local data processing format, increased network bandwidth usage, and improved internal FPDB processing. These optimizations led FPDB to better reflect production-calibur systems and enable our analysis in Chapter 4.

3.1 FlexPushdownDB System Overview

FPDB [25] is an OLAP cloud DBMS. Unlike the other SDA cloud DBMSs mentioned in Section 2.4.1, FPDB can use both caching and computation pushdown at the same time. FPDB has four modes of operation. In the first mode, *pullup*, needed data is always accessed from the storage layer, and all operations are performed in the compute layer. The following two modes build upon the pullup mode by introducing caching with no computation pushdown and computation pushdown with no caching. They are called the *caching-only* and *pushdown-only* modes, respectively. The final mode performs the operations on cached data in the compute layer and computation pushdown on the remaining data and is referred to as the *hybrid* mode.

3.1.1 Environment

FPDB runs on AWS. A single EC2 instance [3] makes up the compute layer. AWS S3 [5] acts as the storage layer. To perform computation pushdown, FPDB uses S3 Select requests [18], enabling pushdown for `filter` and `project` operations. When not performing computation pushdown, FPDB requests the data via S3 GET requests, which transfer an entire file from the storage layer to the compute layer.

3.1.2 Internal Data Format

FPDB operates on data in memory using Apache Arrow [9], which we will refer to as *Arrow format*. Arrow format is a columnar memory format that enables columnar processing within FPDB. This format is advantageous for the read-heavy workloads performed by OLAP DBMSs [22].

3.1.3 Caching

Large tables are often split into chunks when stored in a DBMS, each of these chunks is referred to as a single table partition. FPDB implements a type of table-level caching on a unit called a *segment*. We define a segment in this context as a single column of a single table partition. Given this design, FPDB can cache a subset of columns for a given table partition. FPDB performs in-memory caching and stores cache data in Arrow format. FPDB chooses which data to cache and which to evict based on a user-specified caching policy.

3.1.4 Actor Model

FPDB uses the actor model [12] to achieve high parallelism during query execution. In the context of FPDB, an actor represents a DBMS operation on a single partition of a table. FPDB uses the C++ Actor Framework (CAF) [13] implementation. CAF uses an efficient work-stealing scheduling algorithm to maximize CPU usage.

FPDB represents DBMS operations as a tree. The leaf operators first access a single partition's data from either the cache or the storage, and the root operator

aggregates and outputs the final result. When processing a query, FPDB first constructs the operators as actors. It then starts these actors and uses CAF to manage actors' execution, message passing, and message polling.

3.2 Experimental Settings

For all performance experiments in this chapter, we use the Star Schema Benchmark (SSB) [21] dataset with scale factor 100. We store this data in S3 in uncompressed comma-separated value (CSV) format. In uncompressed CSV format, the total dataset is roughly 64 GB. The lineorder table is partitioned into 16 MB shards, each of which contains 150,000 rows. The lineorder partitions are sorted on their `lo_orderdata` column, which is uniformly distributed over a time span. We evaluate performance with 50 test queries that we randomly generate following the SSB queries' template. When running FPDB in modes that use caching, we use a 6 GB cache, select a least frequently used (LFU) caching policy, and perform 30 queries to warm up the cache before the 50 test queries. We run all experiments on a c5n.9xlarge EC2 instance, which has 25 Gb/s network bandwidth and 36 virtual CPUs.

To control the cache hit rate, we introduce skew to the queries' data access patterns. We add skew to the data access pattern on the lineorder table, which accounts for over 98% of the stored data in uncompressed CSV format. We use a Zipfian [17] distribution for skew, with a choice of parameters resulting in 80% of the data accesses pointing to 20% of the data. For our test queries, this choice of distribution, cache size, and caching policy results in a test-query cache hit rate of approximately 73% for the caching-only and hybrid modes.

3.3 Data Conversion Optimization

When FPDB first receives data in CSV format, it must convert that data to Arrow format for internal processing. The Arrow API enables straightforward conversion from CSV to Arrow format. To do this, FPDB can place the CSV data into a buffer,

pass that buffer to an Arrow InputStream, and then provide the Arrow InputStream to an Arrow CSV TableReader. This Arrow CSV TableReader then converts the data from CSV to Arrow format in user-specified chunk sizes. The final result is an Arrow table containing the table columns, which FPDB can access individually.

S3 GET and Select requests receive results in two different ways that slightly modify their conversion step from CSV to Arrow format. For S3 GET requests, the result is a single stream containing the transferred file. The user can then read from this stream to read the transferred file. For S3 Select requests, the results are received as a series of buffered chunks of data. Each buffered chunk is received over time for the duration of the request until the request is complete.

3.3.1 Initial Implementation

The original FPDB CSV-to-Arrow conversion for S3 GET responses was implemented as follows.

1. Receive the stream containing the transferred file
2. Read 100 kB from the stream into a buffer
3. Create an Arrow InputStream
4. Pass the 100 kB buffer to the Arrow InputStream
5. Create an Arrow CSV TableReader
6. Feed the Arrow InputStream to the Arrow CSV TableReader
7. Copy data from the Arrow Input Stream to the Arrow CSV TableReader
8. Output the copied data in Arrow format using the Arrow CSV TableReader
9. Go to step 7 if there is more data to convert from the Arrow InputStream's 100 kB buffer
10. Go to step 2 if there is more data in the stream containing the transferred file.

This has two significant performance problems. First, this creates two copies of all data. The first copy occurs when the data is read from the stream into the buffer in step 2, and the second is when the Arrow CSV TableReader copies the data from the Arrow InputStream in step 7. Second, an Arrow CSV TableReader object has a large

overhead, so using a new one for each 100 kB chunk slows down the data conversion step. This has a large impact on the pullup and caching-only modes in FPDB, as they receive significantly more data across the network than the pushdown-only and hybrid modes receive.

3.3.2 Improving Initial Implementation

To address both of these issues, we sub-classed the Arrow InputStream with our own implementation that could read from the S3 Get response stream directly. We then passed this Arrow InputStream to a single Arrow CSV TableReader. This removed the copy operation in step 2 and eliminated the overhead associated with constructing and running an Arrow CSV TableReader for each 100 kB response chunk. S3 Select responses are received in a buffer rather than a stream, so the only optimization we could perform for S3 Select CSV-to-Arrow conversion was to share an Arrow CSV TableReader for each buffered chunk received.

This optimization increased the performance of FPDB for the pullup and caching-only modes by **2.12x** and **1.94x**. The impact was negligible for the pushdown-only and hybrid modes, as shown in Figure 3-1. This is because the amount of data returned and converted to Arrow format is an order of magnitude smaller than the other modes due to computation pushdown and the selectivity of the SSB test queries.

3.3.3 Implementing our own CSV-to-Arrow Converter

Despite this improvement, there was another performance issue in the conversion step. The Arrow CSV TableReader first parses CSV data for delimiters by going through the file byte-by-byte. It then uses this information to index into the CSV data and convert it to Arrow format for each column. This method for finding delimiters is inefficient, as only a single byte is processed at a time.

To improve upon the Arrow CSV TableReader's delimiter processing, we built our own CSV-to-Arrow converter. We created this converter to take as input either an input stream from S3 GET or a buffered chunk of data from S3 Select. In either case,

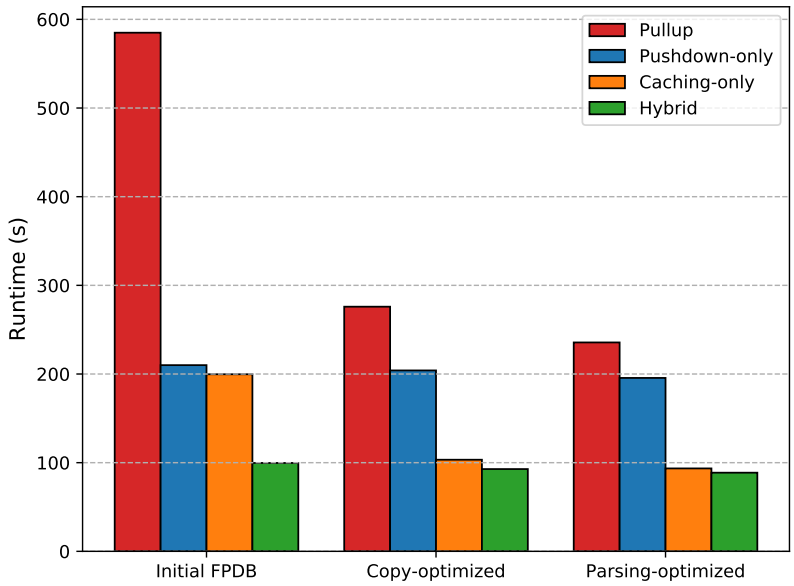


Figure 3-1: The overall runtimes for all modes during the test queries when using the initial CSV-to-Arrow converter, the improved CSV-to-Arrow converter after removing redundant copies and conversion overheads, and our new CSV-to-Arrow converter that uses SIMD instructions to parse the CSV data and converts data column by column.

the converter parses and converts the input in chunks of user-specified size. For the delimiter parsing step, we adapted code from `simdcsv` [16], an extremely fast CSV parser that takes advantage of single input multiple data (SIMD) instructions. This dramatically reduced CSV delimiter parsing time.

After finding the delimiters, we then needed to convert the data to Arrow format. Our initial implementation did this row-by-row, reading in each column in the row, checking the output data type, and converting the CSV data to the corresponding data type.

This method to convert data led to similar performance to the Arrow CSV TableReader. After further profiling and testing, we found that converting the data column-by-column was far more performant. This new parsing and conversion resulted in a **1.17x** speedup for the pullup mode and **1.10x** for the caching mode on top of the performance improvements in Section 3.3.2. This improvement can be seen in Figure 3-1. As in the previous CSV-conversion optimization, pushdown-only and hybrid modes received minimal benefit since they convert significantly less data.

Despite this new converter being almost an order of magnitude faster than the initial implementation, this had a much smaller impact on overall runtime than the optimization in Section 3.3.2 due to other bottlenecks in the system, which we address in Sections 3.4 and 3.5.

3.4 Removing IO-Blocking Operations and Increasing Network Bandwidth Usage

FPDB's actor implementation from Section 3.1.4, CAF [13], has a scheduler that attempts to maximize CPU usage while mitigating slowdowns due to oversubscription of resources. Since CAF is implemented in userspace, it is unable to kill processes. This means that once CAF starts an actor, that actor must run to completion. To maximize CPU usage without oversubscribing resources, the CAF scheduler only runs up to a predetermined number of actors on a machine at any given time. This number is set based on the machine's available resources.

This design works well for compute-intensive tasks but can create problems when IO-blocking operations are introduced, such as S3 GET and Select. If CAF is already running the maximum number of actors and an actor performing an S3 GET request is waiting for a response, every other actor ready to execute must wait. This can reduce CPU utilization in FPDB, as an IO-blocking task may be executed in place of a compute-intensive task. Additionally, the AWS documentation states that sending more concurrent S3 GET or Select requests can increase network bandwidth saturation [8]. This means that limiting the number of concurrent S3 requests can also limit network bandwidth usage. CAF aims to solve problems like this with the concept of *detachment*. A detached actor can run in the background without taking up one of these slots reserved for compute-intensive tasks.

When FPDB was initially implemented, it did not take advantage of detached actors. However, merely modifying the existing system to detach actors that perform S3 GET or Select requests does not solve this problem. This is because the actors

performing S3 GET and Select also handle the conversion to Arrow in FPDB, which is a compute-intensive task. When we initially tried designating all actors with S3 GET or Select requests as detached, this led to oversubscribing the CPUs and poor system performance due to the compute-intensive conversion to Arrow format. To fix this problem while still using detached actors, we introduced a lock to limit how many actors performing S3 GET or Select can perform their conversion step at a time, mitigating oversubscription of CPUs. With this fix, detached actors dramatically improved system performance, as can be seen in Figure 3-2.

This optimization resulted in a slight performance improvement for the FPDB pullup and caching-only modes of **1.13x** and **1.12x**. This change significantly improved the pushdown-only and hybrid modes by **2.67x** and **1.77x** respectively. The pushdown-only and hybrid modes benefitted from this change with dramatically improved network bandwidth usage. Because these modes perform computation pushdown, they could not saturate the network bandwidth when running the previous maximum number of requests. Enabling some detached actors led to running roughly order of magnitude more requests in parallel, which improved resource utilization for these modes and drastically improved runtime. The pullup and caching-only modes were already saturating network bandwidth with relatively few requests, as they do not perform computation pushdown and therefore transfer significantly more data across the network. They only benefitted from the removal of IO-blocking operations that were preventing compute-intensive operations, accounting for their more modest improvement compared to the pushdown-only and hybrid modes.

3.5 Improving Internal Processing

FPDB initially used 16 MB shards for the majority of internal processing. This led to a very high network bandwidth usage after the change in Section 3.4, as many S3 GET and Select requests were now sent in parallel with one request per shard. One drawback of this small shard size is that each operation works with a relatively small amount of data at a time, accessing only a single partition worth of data.

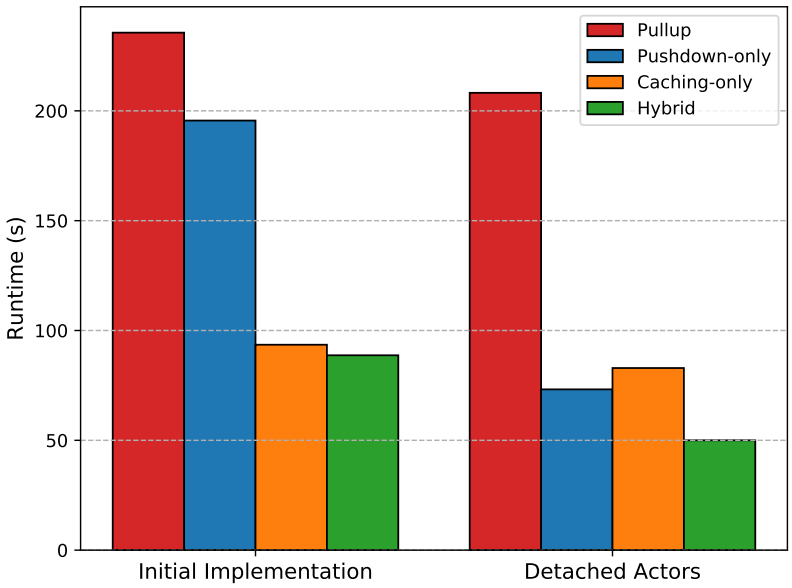


Figure 3-2: The overall test-query runtimes for all modes, before and after detaching actors that perform S3 GET and Select requests. These improvements are from removing IO-blocking operations and using more network bandwidth.

Another minor drawback is that using small shards increased the number of actors in our system, with one actor assigned per shard accessed in a query. This caused more actors to be used for each query, leading to higher system overheads from more message passing and context switches.

We changed the partition size for the lineorder table data from 16 MB to 160 MB to address both of these problems. This change resulted in significantly faster local processing in our system. Initially, we only sent one S3 GET or Select request at a time for each partition. This resulted in 10x fewer partitions and 10x fewer requests, reducing our network utilization since S3 can better saturate the network bandwidth when responding to more requests. Even though our local processing was now faster, this considerable reduction in network bandwidth usage slowed down overall runtime for all modes, as shown in Figure 3-3.

To address the reduction in network bandwidth while still working with larger batches internally in FPDB, we modified our code to leverage the S3 GET and Select byte-range scan functionality. With a byte-range scan, a subset of a partition can be selected by a request. This allowed us to break up a single 160 MB request

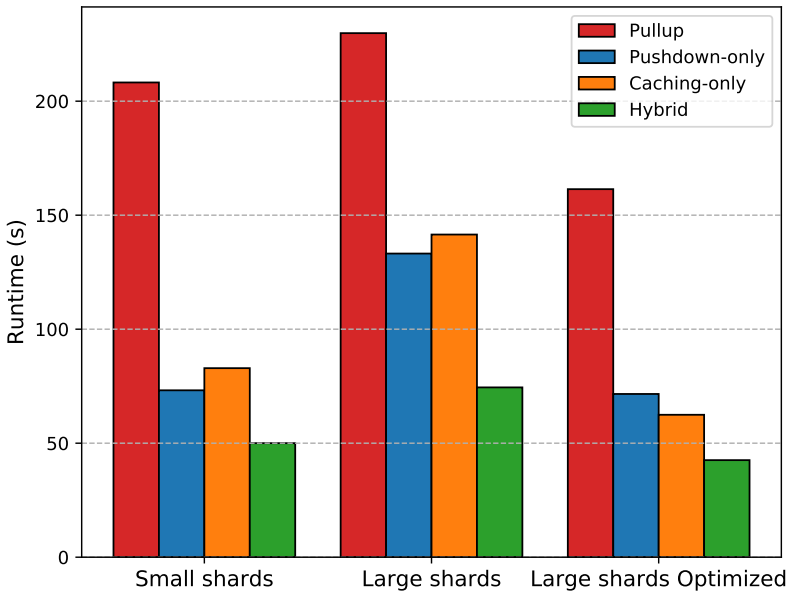


Figure 3-3: The overall runtimes for all modes executing the test queries when using 16 MB shards for internal processing, 160 MB shards for internal processing, and finally 160 MB shards with byte-range scan. Using larger shards increases FPDB local processing speed but initially reduced network bandwidth due to sending fewer requests. Once we used byte-range scans to increase the number of S3 requests, the benefit of using larger shard sizes for internal processing can be seen.

into ten 16 MB requests and achieve our previous network bandwidth usage while still maintaining the local processing benefits of larger partitions. The impact of this change is shown in Figure 3-3. Once using byte-range scans, this optimization resulted in a speedup of over **1.2x** for the pullup, caching-only, and hybrid modes. Our internal filter operation benefited the most from this change while only lightly impacting other operators, so pushdown-only was only minorly improved.

With the optimizations contributed in this chapter, we felt that FPDB more accurately reflected a production-caliber system with bottlenecks stemming from the constraints imposed by a cloud DBMS using an SDA. The network bottleneck in an SDA poses the question of how much computation pushdown most reduces runtime for a DBMS in this setting. In the next chapter, we introduce a model that predicts runtime when using the caching-only, pushdown-only, or hybrid mode, and validate

its performance using experiments on FPDB. We further use this model to predict runtime in diverse experimental settings, including varied network bandwidth, local processing speed, and computation pushdown speed.

Chapter 4

SDA Runtime Model

This chapter defines a runtime model for cloud DBMSs that use an SDA. The model predicts the runtime for queries that can be performed via computation pushdown. Specifically, we examine `filter` operations on a single integer column using the SSB [21] dataset and the same experimental settings as Chapter 3. We leave modelling the runtime of query operations such as `join` and aggregation functions to future work.

We first define the model and the individual system components that contribute to the overall runtime. We next design and run experiments to estimate the different model parameters when instantiated for FlexPushdownDB (FPDB) [25] running on AWS with an EC2 instance [3] as the compute layer and S3 [5] as the storage layer. We then validate the model using these experimentally-gathered parameters for the caching-only, pushdown-only, and hybrid modes of FPDB. Finally, we use the model to predict both the runtime and optimal mode of operation under various settings. We aim to determine how much computation pushdown results in the optimal runtime for a cloud DBMS using an SDA.

4.1 Model Definition

The overall runtime for `filter` in a cloud DBMS can be abstracted into a combination of 3 different components.

1. $T_{storage}$: The time to perform filter operations on data in the storage layer.

2. $T_{transfer}$: The time to transfer data from the storage layer to the compute layer.
3. $T_{compute}$: The time to perform filter operations on data in the compute layer.

These are abstract components that we expect to be applicable to most systems. This section explains the relationship between these components and the runtime for each FPDB mode of operation.

4.1.1 Caching-Only

We estimate the runtime for the FPDB caching-only mode to be

$$T_{caching} = T_{transfer} + T_{compute}. \tag{4.1}$$

Here $T_{transfer}$ and $T_{compute}$ are as defined in Section 4.1. We sum these two runtime components because they are performed serially in FPDB. We do not include the time to convert the data from the storage format to the compute format, as this step can be performed in parallel with the data transfer step. The conversion rate also tends to be significantly faster than the transfer rate in an optimized system such as FPDB, so the conversion time is insignificant when compared to the transfer time.

4.1.2 Pushdown-Only

For the pushdown-only mode we estimate runtime to be

$$T_{pushdown} = T_{storage} + T_{transfer}. \tag{4.2}$$

We add the time to perform storage processing and the time to transfer data because these steps appear to be sequential in S3, as we will show in Section 4.3.2. As in the caching-only mode, data can be converted in parallel to the data transfer, so data conversion time does not appear in (4.2).

4.1.3 Hybrid

Drawing from the caching-only and pushdown-only modes, we estimate the hybrid mode's runtime to be

$$\begin{aligned} T_{hybrid} &= \max(T_{compute}, T_{storage} + T_{transfer}) \\ &= \max(T_{compute}, T_{pushdown}). \end{aligned} \tag{4.3}$$

For the hybrid mode, $T_{compute}$ is the time to perform the computation on cached data in the compute layer, and $T_{pushdown}$ is the time in (4.2) that it takes to perform computation pushdown on the rest of the data. We use the maximum of these runtime components, since the compute-layer processing and computation pushdown use different compute resources and can therefore be fully parallelized.

4.2 Model Inputs and Parameters

With the runtime components defined in Section 4.1, we now define the inputs and parameters used to compute each of these partial runtimes and how they combine to form the overall runtime. The inputs are

- S: the size of the data in storage format
- R: the ratio of the storage data format size to the compute data format size
- h: cache hit ratio. For pushdown-only mode, $h = 0$ as FPDB performs no caching in this mode
- r: row selectivity of the query (percentage of rows selected)
- c: column selectivity of the query (ratio of bytes in the selected columns / all columns)
- $BW_{network}$: the network bandwidth between the compute and storage layers listed by AWS,

and the parameters are

- $BW_{compute}$: compute layer query processing speed
- $BW_{storage}$: storage layer query processing speed

- $BW_{network-achieved}$: The achieved network bandwidth between the compute and storage layers
- $T_{compute-overhead}$: compute layer query processing overhead
- $T_{storage-overhead}$: storage layer query processing overhead.

The following sections describe how we calculate $T_{storage}$, $T_{transfer}$, and $T_{compute}$ in FPDB running on an EC2 instance and using S3 to store our dataset described in Section 3.2, which is stored in uncompressed CSV format.

4.2.1 Storage-Processing Time

Storage processing time is the time to perform computation pushdown on data that is not in the cache. For `filter` operations performed in S3, we approximate this as a function of the amount of data scanned and the compute pushdown processing speed. We also add a component to account for FPDB's and S3's overhead for S3 Select requests. This leads to the computation pushdown time

$$T_{Storage} = \frac{(1-h)S}{BW_{storage}} + T_{storage-overhead}. \tag{4.4}$$

4.2.2 Data Transfer Time

Data transfer time is the time to transfer data from the storage layer to the compute layer over the network. We approximate this as a function of the amount of data returned from the storage layer and the achieved network bandwidth between the compute and storage layers. When **not** performing computation pushdown, as in the caching-only mode of FPDB, all data that is not present in the cache must be transferred. When performing computation pushdown, only a subset of the data is returned depending on the queries' row and column selectivities.

$$T_{transfer} = \begin{cases} \frac{(1-h)S}{BW_{network-achieved}} & \text{, if caching-only} \\ \frac{(1-h)rcS}{BW_{network-achieved}} & \text{, otherwise.} \end{cases} \tag{4.5}$$

4.2.3 Compute-Processing Time

We initially thought that `filter` processing in FPDB would be a function of only the amount of data filtered (S) and the compute layer processing speed ($BW_{compute}$). However, when running experiments, we found that this was not to be the case. FPDB uses Apache Gandiva [10] internally for filter operations. Gandiva uses LLVM [19] for filter generation and splits filters into a *filter* and a *project* step, where the project step outputs the final result. We found that the project step dominates runtime for FPDB's filter operation, and the project step's runtime scales with the filter output size. This led us to approximate $T_{compute}$ as

$$T_{compute} = \begin{cases} \frac{rc\frac{S}{R}}{BW_{compute}} + T_{compute-overhead} & \text{, if caching-only} \\ \frac{hrc\frac{S}{R}}{BW_{compute}} + T_{compute-overhead} & \text{, if hybrid,} \end{cases} \tag{4.6}$$

where $\frac{S}{R}$ is the size of the data in the compute data format, and rc estimates the total percent of the data that is output. The hybrid mode performs processing on cached data in the compute layer and computation pushdown on the remaining data, so for the hybrid mode this term includes the cache hit ratio (h).

4.2.4 Model with Inputs and Parameters

Using the inputs and parameters defined in Section 4.2 for FPDB running on AWS, our complete model for each mode is

$$\begin{aligned} T_{caching} &= T_{transfer} + T_{compute} \\ &= \frac{(1-h)S}{BW_{network-achieved}} + \frac{rc\frac{S}{R}}{BW_{compute}} + T_{compute-overhead} \end{aligned} \tag{4.7}$$

$$\begin{aligned} T_{pushdown} &= T_{storage} + T_{transfer} \\ &= \frac{(1-h)S}{BW_{storage}} + T_{storage-overhead} + \frac{(1-h)rcS}{BW_{network-achieved}} \end{aligned} \tag{4.8}$$

$$
\begin{aligned}
T_{hybrid} &= \max(T_{compute}, T_{storage} + T_{transfer}) \\
&= \max(T_{compute}, T_{pushdown}) \\
&= \max(\frac{hrc\frac{S}{R}}{BW_{compute}} + T_{compute-overhead}, \\
&\qquad \frac{(1-h)S}{BW_{storage}} + T_{storage-overhead} + \frac{(1-h)rcS}{BW_{network-achieved}}).
\end{aligned} \tag{4.9}
$$

4.3 Calibrating Bandwidths

The inputs in Section 4.2, such as S, r, c, are determined by the query, and other inputs such as $BW_{network}$ are determined by the specifications of the machines representing the compute and storage layers. The parameters $BW_{compute}$, $BW_{storage}$, $BW_{network-achieved}$, $T_{compute-overhead}$, and $T_{storage-overhead}$ are implementation-specific and need to be learned for the compute and storage layers: FPDB and S3 in our case.

4.3.1 Learning $\mathbf{BW_{compute}}$ and $\mathbf{T_{compute-overhead}}$

To calculate $BW_{compute}$ and $T_{compute-overhead}$ for FPDB, we ran a series of experiments in FPDB on fully cached data. We varied the number of bytes filtered (S) and the selectivity of the filter (rc). The experiment settings we tested were

- S = 4.4 GB, 8.7 GB, and 17.0 GB (7%, 14%, and 28% of lineorder table)
- r = 0.14, 0.28, 0.42, 0.56, 0.70, 0.84, 1.0
- c = 0.11, 0.38, 0.53, 1.0 (3, 8, 12, 17 columns),

when filtering on a single integer column.

We ran 20 trials for each configuration on the c5a.8xlarge (10 Gb/s, 32 virtual CPUs) and c5.16xlarge (20 Gb/s, 64 virtual CPUs) instances. The data is noisy because these experiments were run in a cloud environment on shared resources. To reduce the impact of noise on our estimates, we only consider the median runtime for each configuration in our runtime prediction model.

To estimate $T_{compute-overhead}$ and $BW_{compute}$ we ran the following linear regression

on the c5a.8xlarge and c5a.16xlarge instances.

$$T_{compute}(r, c, S, R) = \frac{rc\frac{S}{R}}{BW_{compute}} + T_{compute-overhead}, \tag{4.10}$$

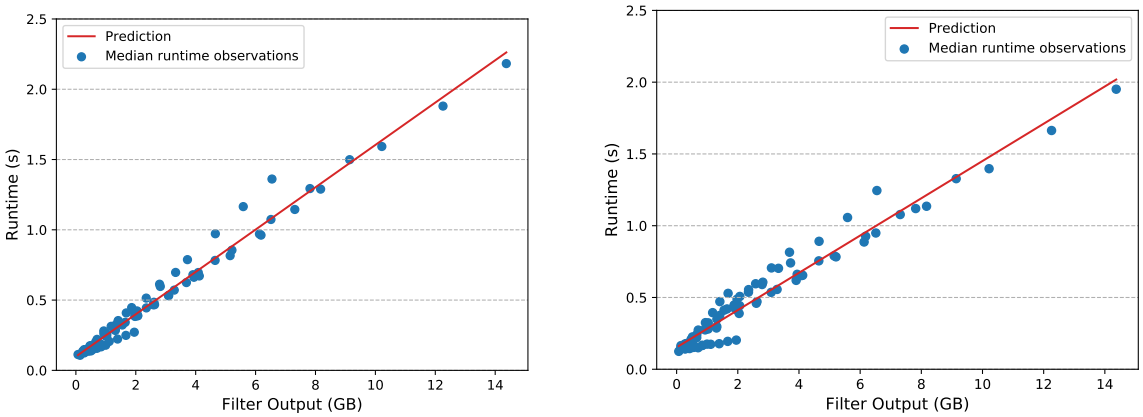
where r, c, S, and R were known values from our dataset and test queries, and $rc\frac{S}{R}$ is the filter output size. We chose to use linear regression as it has a low memory footprint, is lightweight, and is easy to interpret [20]. Our regression estimated $T_{compute-overhead}$ to be 0.10 and 0.15 seconds and $BW_{compute}$ to be 6.64 and 7.70 GB/s for the c5a.8xlarge and c5a.16xlarge instances respectively. The linear regression for both instances is shown in Figures 4-1a and 4-1b.

We chose to use the the coefficient of determination r^2 to evaluate the learned linear regression since it is a common metric for regression evaluation in machine learning. The r^2 value of a regression is defined as

$$r^2 = 1 - \frac{\sum_{i=1}^{n}(Actual_i - \overline{Actual})^2}{\sum_{i=1}^{n}(Actual_i - Predicted_i)^2}, \tag{4.11}$$

where n is the number of tested configurations, $Actual_i$ is the ith observed data point, $Predicted_i$ is the ith predicted datapoint, and $\overline{Actual}$ is the average observed runtime. The r^2 value of a regression measures how much of the observed data's variance is captured by the learned regression. A model that always predicts the mean observed runtime will have an r^2 value of 0 (barring perfectly uniform datasets), and a model that accurately predicts every observed value will have an r^2 of 1. A larger value indicates a better fit to the data. Our learned linear regression had an r^2 value of **0.980** and **0.954** for the c5a.8xlarge and c5a.16xlarge instances respectively.

In these results, something that stood out was that the filter rate did not come close to doubling despite the doubling of computational resources between the two machines. We speculate that this is because FPDB is unable to maximize parallelism in the system. This does not impact the work that we present in this chapter, however, so we do not pursue this question further here.



(a) c5a.8xlarge (b) c5a.16xlarge

Figure 4-1: The results of the compute layer processing experiments in Section 4.3.1 and the learned linear regression instance on a (a) c5a.8xlarge and (b) c5a.16xlarge instance.

4.3.2 Calibrating $\mathbf{BW_{storage}}$, $\mathbf{BW_{network-achieved}}$, and $\mathbf{T_{storage-overhead}}$

We ran a series of experiments in FPDB in the pushdown-only mode to estimate $BW_{storage}$ and $T_{storage-overhead}$ for S3 Select, and the ratio $\frac{BW_{network}}{BW_{network-achieved}}$ between the compute and storage layers. We varied the number of bytes filtered (S) and the selectivity of the filter (rc). In (4.2), we define $T_{pushdown}$ as the sum of $T_{storage}$ and $T_{transfer}$, but we cannot isolate $T_{transfer}$ in a simple way, so we varied the network bandwidth by using the c5a.8xlarge and c5a.16xlarge machines as in Section 4.3.1 to set up a system of equations and solve for $BW_{storage}$, $T_{storage-overhead}$, and the ratio $\frac{BW_{network}}{BW_{network-achieved}}$. We tested two experimental settings: one using queries with low selectivities (rc) and one using high selectivities. We chose to use these settings as they have very different amounts of total filtered and returned data, and we wanted to get an estimate that is consistent with the behavior in both cases. The settings for queries with low selectivities that we gathered data for were

- $BW_{network}$ = 10 Gb/s, 20 Gb/s
- S = 4.4 GB, 8.7 GB, and 17.0 GB (1/14, 1/7, 2/7 of lineorder table)
- r = 0.02, 0.2, 0.4, 0.6, 0.8, and 1.0
- c = 0.03, 0.12, 0.14, and 0.21 (1, 2, 3, and 4 columns).

The settings for queries with high selectivities that we gathered data for were

- $BW_{network}$ = 10 Gb/s, 20 Gb/s
- S = 4.4 GB, 8.7 GB, and 17.0 GB (1/14, 1/7, 2/7 of lineorder table)
- r = 0.14, 0.28, 0.42, 0.56, 0.70, 0.84, 1.0
- c = 0.11, 0.38, 0.53, 1.0 (3, 8, 12, and 17 columns).

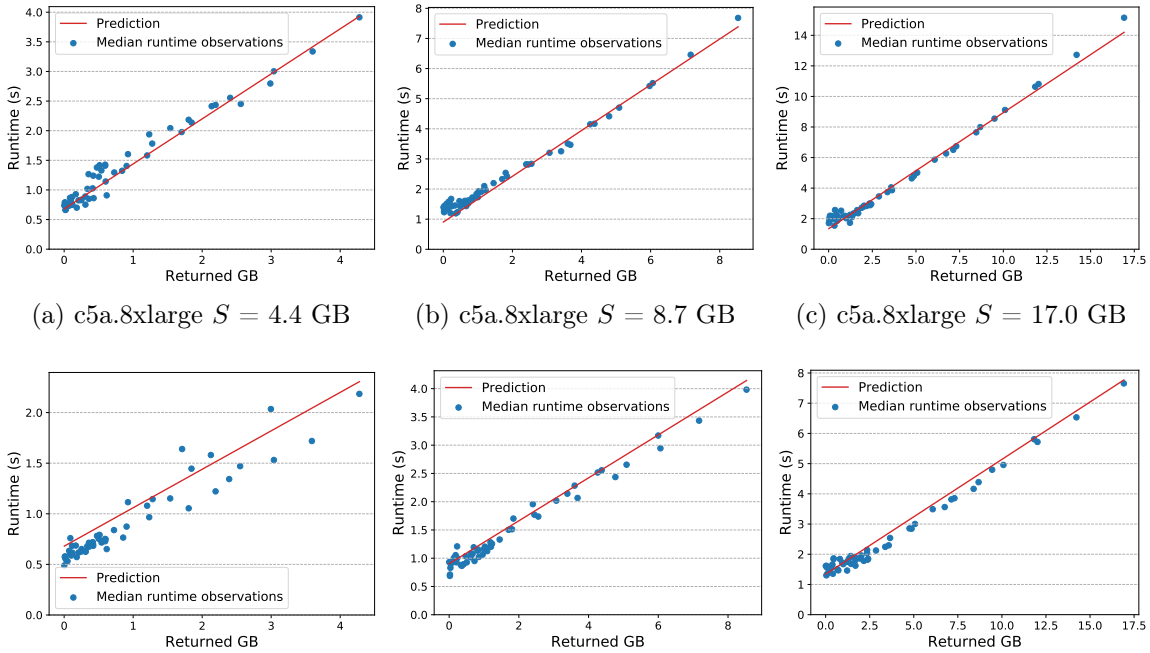
We ran 20 trials at each configuration on the c5a.8xlarge and c5.16xlarge instances. As in 4.3.1, we only consider the median runtime at each configuration when training the model.

We then performed the following linear regression to estimate $T_{storage-overhead}$, $BW_{storage}$, and $\frac{BW_{network}}{BW_{network-achieved}}$.

$$\begin{aligned} T_{pushdown}(r, c, S) &= T_{storage} + T_{transfer} \\ &= \frac{S}{BW_{storage}} + T_{storage-overhead} + (\frac{BW_{network}}{BW_{network-achieved}})(\frac{rcS}{BW_{network}}), \end{aligned} \tag{4.12}$$

where r, c, and S were known based on the test queries. As in Section 4.3.1, we chose to use linear regression because of its low memory and computational requirements, making it practical in a system that receives queries online.

Since we divided the bytes returned (rcS) by $BW_{network}$, we can perform a single linear regression on the data from both instances to find $\frac{BW_{network}}{BW_{network-achieved}}$, despite their different $BW_{network}$ values. The linear regression on this data had an r^2 value of **0.985**. This regression estimated a $T_{storage-overhead}$ of 0.45 seconds and a S3 Select $BW_{storage}$ of 19.1 GB/s. This value for $BW_{storage}$ is higher than $BW_{compute}$, as we can leverage S3's resources to achieve a higher degree of parallelism by sending many S3 Select requests in parallel. The regression estimated $\frac{BW_{network}}{BW_{network-achieved}}$ to be 0.95, leading to a $BW_{network-achieved}$ of 10.5 Gb/s and 21.0 Gb/s for the c5a.8xlarge and c5a.16xlarge instances compared to the true values 10 Gb/s and 20 Gb/s respectively, indicating that the estimated achieved network bandwidth slightly exceeded the EC2 specifications for these machines. The linear regression results on both instances are shown in Figure 4-2.



(a) c5a.8xlarge $S = 4.4$ GB (b) c5a.8xlarge $S = 8.7$ GB (c) c5a.8xlarge $S = 17.0$ GB

(d) c5a.16xlarge $S = 4.4$ GB (e) c5a.16xlarge $S = 8.7$ GB (f) c5a.16xlarge $S = 17.0$ GB

Figure 4-2: The results of the pushdown processing experiments in Section 4.3.2 and the linear regression on the c5a.8xlarge and c5a.16xlarge instance for each S value tested.

4.4 Experimental Validation of Model

After quantifying the parameters for FPDB on the c5a.8xlarge and c5.16xlarge instances using S3, we can now validate our model defined in Section 4.2.4. We validate the pushdown-only mode with a separate set of configurations than the caching-only and hybrid modes, as it does not perform caching. For pushdown-only, we validate the model with the following configurations.

- Instances: c5a.8xlarge, c5a.16xlarge (changing $BW_{compute}$ and $BW_{network-achieved}$)
- $BW_{network-achieved}$ = 10.5 Gb/s and 21.0 Gb/s
- S = 8.7 GB and 17.0 GB (1/7 and 2/7 of lineorder table)
- r = 0.04, 0.08, 0.14, 0.20, 0.40, 0.70, 1.0
- c = 0.11, 0.38, 0.53, 1.0 (3, 8, 12, and 17 columns)
- h = 0.

For the caching-only and hybrid modes of FPDB, we validate with the following configurations.

- Instances: c5a.8xlarge, c5a.16xlarge (changing $BW_{compute}$ and $BW_{network-achieved}$)
- $BW_{network-achieved}$ = 10.5 Gb/s and 21.0 Gb/s
- S = 8.7 GB and 17.0 GB (1/7 and 2/7 of lineorder table)
- r = 0.10, 0.30, 0.60, 1.0
- c = 0.11, 0.38, 0.53, 1.0 (3, 8, 12, and 17 columns)
- h = 0, 0.25, 0.5, 0.75, 1.0.

We chose these varying inputs and parameters to validate the different modes in our model to test the model's robustness and accuracy as these inputs and parameters change. If the model is a good fit for the data, we expect it to fit well across the input and parameter space.

We ran 10 trials for each configuration and used the median runtime as the observed value. Unlike the experiments in Section 4.3, we also vary the cache hit ratio (h) in these experiments for the caching-only and hybrid modes to evaluate the accuracy of our model in the presence of caching.

Table 4.1: MAPE and median APE along with the 80th and 90th percentile APE for each FPDB mode in validation experiments when using the median result in each configuration as the observed value.

Mode	MAPE	Median APE	80th percentile APE	90th percentile APE
Caching-only	10.144%	10.426%	13.072%	14.317%
Pushdown-only	7.901%	6.120%	12.491%	18.547%
Hybrid	13.458%	8.98%	23.111%	32.568%

We used the mean absolute percent error (MAPE) to evaluate our model's accuracy, defined as

$$MAPE = 100\% \times \frac{1}{n} \sum_{i=1}^{n} \frac{|Actual_i - Predicted_i|}{Actual_i}, \tag{4.13}$$

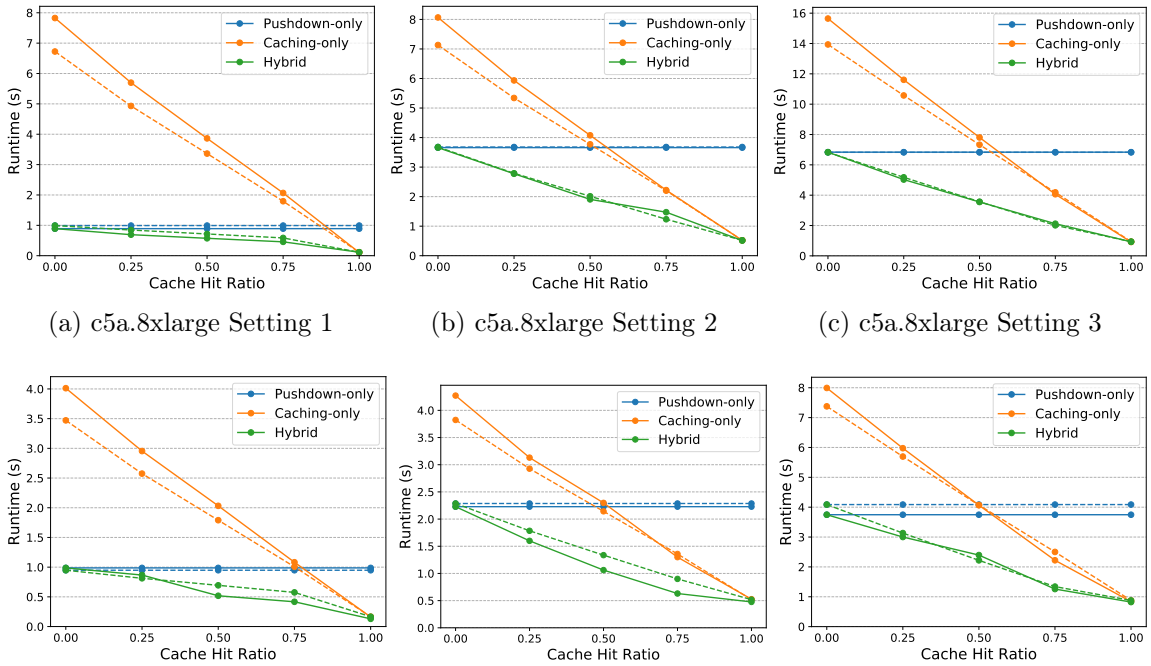
where n is the number of tested configurations. We calculated the MAPE and median APE, along with the 80th and 90th percentile APEs for each mode in FPDB. These values are reported in Table 4.1.

The hybrid mode has the highest error of all the modes in the validation settings. When the hybrid mode has a high cache hit rate and only queries a small amount of data from S3, the runtime is highly variable and harder to predict. We believe that this accounts for the higher error rates for the hybrid mode, and that the noise from S3 Select also results in the higher 90th percentile APE for the pushdown-only mode.

We will also show the predicted runtimes and observed runtimes for three different configurations on each machine as the cache hit ratio varies to show the model's robustness as the amount of data scanned and filter selectivity change. The settings we show are

1. S=8.7 GB, r=0.1, c=0.11 (1 column)
2. S=8.7 GB, r=0.6, c=0.53 (12 columns)
3. S=17.0 GB, r=0.6, c=0.53 (12 columns).

The results of our model on these settings for the c5a.8xlarge and c5a.16xlarge instance are shown in Figure 4-3. As shown in the figure and in Table 4.1, the model



(a) c5a.8xlarge Setting 1 (b) c5a.8xlarge Setting 2 (c) c5a.8xlarge Setting 3

(d) c5a.16xlarge Setting 1 (e) c5a.16xlarge Setting 2 (f) c5a.16xlarge Setting 3

Figure 4-3: The observed runtimes and the predicted runtimes of the validation experiments on each instance with varying cache hit ratios, using the 3 settings described in Section 4.4. The solid lines show the observed runtime values, while the dashed lines are the predictions of our model.

is very accurate and has an average APE of less than 8% for the caching-only and pushdown-only modes, and an average APE of less than 13.5% for the hybrid mode.

4.5 Model Interpretation

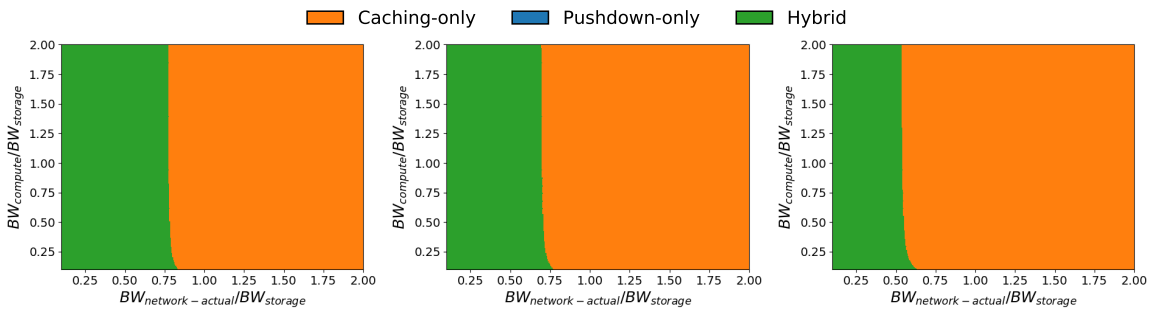
When caching and computation pushdown are available, there are three ways to process a query for a cloud DBMS using an SDA. The first is to process all cached data in the compute layer and transfer the remaining data from the storage layer to the compute layer for processing, represented by the FPDB caching-only mode. The second is to process all cached data in the compute layer and perform computation pushdown on the rest of the data, represented by the FPDB hybrid mode. The last option is to perform computation pushdown on all data, regardless of the cache contents, represented by the FPDB pushdown-only mode.

Table 4.2: Experimental settings for model predictions as the parameter bandwidths vary.

	S	r	c
Low Selectivity	15 GB	0.1	0.1
Medium Selectivity	15 GB	0.5	0.5
High Selectivity	15 GB	0.8	0.8

Using our model, we aim to show which mode of operation is most performant as $BW_{network-achieved}$, $BW_{compute}$, $BW_{storage}$, and h (cache hit ratio) vary. We examine how the different modes perform under queries with low, medium, and high selectivities (rc) with the three settings listed in Table 4.2.

In these experiments we vary the ratio of $\frac{BW_{compute}}{BW_{storage}}$ and $\frac{BW_{network-achieved}}{BW_{storage}}$ from 0.1 to 2.0. We set $BW_{storage}$ fixed at 20 GB/s and vary the other bandwidths accordingly to get the desired ratios. We test cache hit ratios of 0.25, 0.50, and 0.75. We do not consider a cache hit ratio of 0, as it would result in the same runtime for the pushdown-only and hybrid modes, since they will both perform computation pushdown on all data. Similarly, we do not examine a cache hit ratio of 1.0 as it would result in the same runtime for the caching-only and hybrid modes, which will process all data in the compute layer. We use the fixed overheads for the compute layer processing and for computation pushdown that were computed in Section 4.3. While the precise values for $T_{compute-overhead}$ and $T_{storage-overhead}$ for FPDB and S3 may not generalize to all systems, most systems likely encounter a similar overhead during these phases of processing, so we include this time. For the ratio of data storage format size to compute data format size R, we chose to use $R = 1.3$, which we found to be representative of our SSB dataset [21] using uncompressed CSV for the storage layer data format and Arrow [9] for the compute layer data format. This value is relatively low, as many of the integer fields in our dataset are small and require more bytes to be represented as integers in the compute data format than as a sequence of characters in the storage data format.



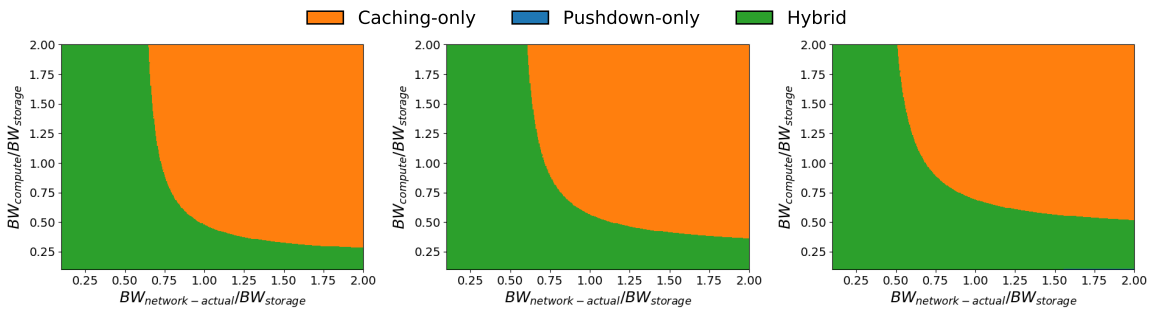
(a) $h = 0.25$ (b) $h = 0.50$ (c) $h = 0.75$

Figure 4-4: The projected optimal mode under the low-selectivity experimental setting with varying bandwidth ratios.

4.5.1 Low Selectivity

The mode that our model predicts to be optimal for varying bandwidths in the low-selectivity setting is shown in Figure 4-4. In this setting, only the hybrid and caching-only modes are ever predicted to be optimal. The pushdown-only mode is estimated to perform worse than the hybrid mode because the compute layer processing is significantly faster than the storage layer processing when the filter selectivity is this low. This difference in runtime is because FPDB's processing is a function of the filter selectivity (rc) and S3's processing is a function of the input data size (S), as shown in (4.7). Hence, a low filter selectivity favors the hybrid mode.

There is a decision point between the caching-only and hybrid modes that occurs at lower $\frac{BW_{network-achieved}}{BW_{storage}}$ ratios as the cache hit ratio increases. This occurs as the caching-only mode's transfer time is much greater than the hybrid mode's transfer time when $BW_{network-achieved}$ is very low. This is because the hybrid mode returns significantly less data than the caching-only mode. As the $BW_{network-achieved}$ increases, the transfer time decreases for both modes, and since the compute layer processing is faster than the storage layer processing in this query setting, the caching-only mode outperforms the hybrid mode.



(a) $h = 0.25$ (b) $h = 0.50$ (c) $h = 0.75$

Figure 4-5: The projected optimal mode under the medium-selectivity experimental setting with varying bandwidth ratios.

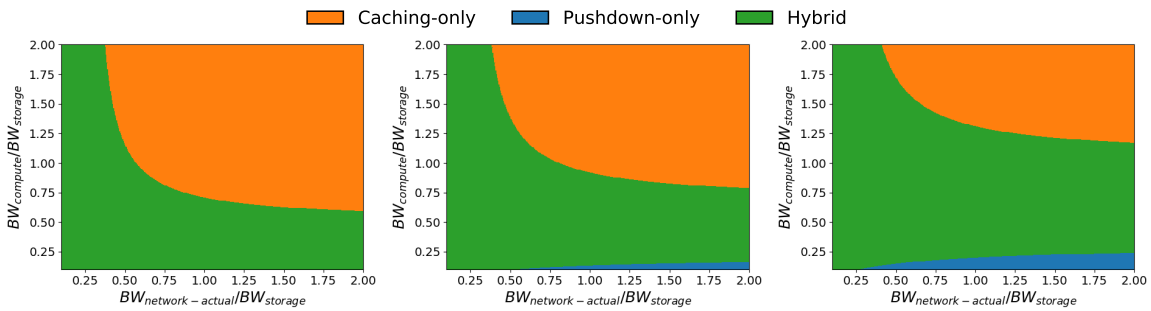
4.5.2 Medium Selectivity

The mode that our model predicts to be optimal for varying bandwidths in the medium-selectivity setting is shown in Figure 4-5. As in the low-selectivity setting, hybrid mode always outperforms pushdown-only mode, as the overall compute layer processing time is still faster than the storage layer processing time at this selectivity and these cache hit ratios. Again, there is also a division between the hybrid and caching-only modes as the $BW_{network-achieved}$ increases, because when $BW_{network-achieved}$ is low the caching-only mode has a relatively longer transfer time, causing it to be slower than the hybrid mode.

These results differ from the low-selectivity setting as the hybrid mode now outperforms the caching-only mode when $BW_{compute}$ is relatively low and $BW_{network-achieved}$ is relatively high. This is because the caching-only mode's compute layer processing time increases with the query selectivity, so the hybrid mode can outperform the caching-only mode when there is a relatively low $BW_{compute}$. When combined with the higher filter selectivity in this setting, the caching-only mode is therefore slower than the hybrid mode.

4.5.3 High Selectivity

Lastly, for the high-selectivity setting, the model's predicted optimal mode is shown in Figure 4-6. These results are very similar to the medium-selectivity setting, with



(a) $h = 0.25$ (b) $h = 0.50$ (c) $h = 0.75$

Figure 4-6: The projected optimal mode under the high-selectivity experimental setting with varying bandwidth ratios.

one key distinction. The pushdown-only mode is now optimal when $BW_{compute}$ is relatively low and the cache hit ratio is high. This is because a higher selectivity leads to a higher compute layer processing time in the hybrid mode. Therefore, as the $BW_{network-achieved}$ increases, a higher $BW_{compute}$ is required for the hybrid mode to outperform the pushdown-only mode. While a higher selectivity does mean that pushdown-only needs to transfer more data across the network, this bottleneck is offset as $BW_{network-achieved}$ increases, which is why the pushdown-only region grows as this parameter increases. When the cache hit ratio is very low (e.g., 0.25), the storage layer processing time is higher than the compute layer processing time, and so the hybrid mode remains faster than the pushdown-only mode at all bandwidths in Figure 4-6a.

These experiments show that the amount of computation pushdown that minimizes runtime for a cloud DBMS using an SDA varies. It depends on the bandwidths and selectivity of the operations performed via computation pushdown. Therefore, using a single mode of operation for all queries in all settings is not optimal. The amount of computation pushdown to perform should instead be determined at the query level for a cloud DBMS using an SDA based on the system bandwidths, the cache hit rate, and the query's selectivity.

Chapter 5

Conclusion

In this work, we optimized the runtime of FlexPushdownDB (FPDB) [25], a cloud DBMS using an SDA. We further defined a runtime model for DBMSs that use both caching and computation pushdown of `filter` operations. We validated the model's accuracy using FPDB and S3. This work shows that using a single static approach of either

1. Transferring data **not** present in the cache to the compute layer and performing all query processing in the compute layer (i.e., FPDB caching-only mode),
2. Performing query processing in the compute layer for data in the cache and computation pushdown on the remaining data (i.e., FPDB hybrid mode), or
3. Performing computation pushdown on all data, regardless of the cache's contents (i.e., FPDB pushdown-only mode)

is nonoptimal for a DBMS using an SDA, and the query execution mode should be determined by the environment and the cache hit ratio on a per-query basis.

This work focused on optimizing runtime, but computation pushdown and compute layer processing often incur different costs. For example, computation pushdown of all queries tends to be more expensive than local processing of all queries in S3 [25]. Although we did not explore the cost tradeoff of these different approaches in this work, the cost should be taken into consideration when determining a cost-optimal query execution plan. We leave this research direction to future work.

Furthermore, we only examined the tradeoffs between different query execution modes using uncompressed CSV data. It is possible that using other data formats, such as Parquet [7], and incorporating compression techniques on the storage data may result in different trends.

We also only examined this model on `filter` operations. Join processing with computation pushdown would also likely benefit from this analysis, though we expect that a runtime model for join processing would be more complex than the model presented in Chapter 4. We feel that these areas are interesting and valuable avenues that future work should explore.

Bibliography

[1] Alluxio - Data Orchestration for the Cloud.

[2] Amazon Athena - Serverless Interactive Query Service - Amazon Web Services.

[3] Amazon Elastic Compute Cloud.

[4] Amazon Redshift.

[5] Amazon S3.

[6] Amazon Web Services (AWS) - Cloud Computing Services.

[7] Apache Parquet.

[8] Performance Design Patterns for Amazon S3 - Amazon Simple Storage Service.

[9] Apache Arrow, 2016.

[10] Gandiva: A LLVM-based Analytical Expression Compiler for Apache Arrow, December 2018.

[11] Presto, 2018.

[12] Gul Abdulnabi Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. June 1985. Accepted: 2004-10-20T20:10:20Z.

[13] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures*, 45:105–131, April 2016. arXiv: 1505.07368.

[14] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, San Francisco California USA, June 2016. ACM.

[15] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. pages 249–264, 2016.

[16] geofflangdale. simdcsv, April 2021. original-date: 2019-04-29T23:25:06Z.

[17] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, SIGMOD '94, pages 243–252, New York, NY, USA, May 1994. Association for Computing Machinery.

[18] Randall Hunt. S3 Select and Glacier Select – Retrieving Subsets of Objects, November 2017. Section: Amazon S3 Glacier.

[19] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, page 75, USA, March 2004. IEEE Computer Society.

[20] Qingzhi Ma, Ali M Shanghooshabad, Mehrdad Almasi, Meghdad Kurmanji, and Peter Triantafillou. Learned Approximate Query Processing: Make it Light, Accurate and Fast. *CIDR*, page 11, 2021.

[21] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The Star Schema Benchmark and Augmented Fact Table Indexing. pages 237–252, August 2009.

[22] M. Stonebraker, D. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, M. Ferreira, Edmond Lau, Amerson Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, Nga Tran, and S. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.

[23] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, Chicago Illinois USA, May 2017. ACM.

[24] Ronald Weiss. A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server, 2012.

[25] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. 2021.

[26] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. PushdownDB: Accelerating a DBMS Using S3 Computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1802–1805, Dallas, TX, USA, April 2020. IEEE.