PushdownDB: Accelerating a DBMS Using S3 Computation

Xiangyao Yu*, Matt Youill‡, Matthew Woicik†, Abdurrahman Ghanem§,
Marco Serafini¶, Ashraf Aboulnaga§, Michael Stonebraker†
*University of Wisconsin-Madison †Massachusetts Institute of Technology
‡Burnian §Qatar Computing Research Institute ¶University of Massachusetts Amherst
Email: yxy@cs.wisc.edu, matt.youill@burnian.com, mwoicik@mit.edu, abghanem@hbku.edu.qa,
marco@cs.umass.edu, aaboulnaga@hbku.edu.qa, stonebraker@csail.mit.edu

***Abstract*—This paper studies the effectiveness of pushing parts of DBMS analytics queries into the Simple Storage Service (S3) of Amazon Web Services (AWS), using a recently released capability called S3 Select. We show that some DBMS primitives (filter, projection, and aggregation) can always be cost-effectively moved into S3. Other more complex operations (join, top-K, and group-by) require reimplementation to take advantage of S3 Select and are often candidates for pushdown. We demonstrate these capabilities through experimentation using a new DBMS that we developed, *PushdownDB*. Experimentation with a collection of queries including TPC-H queries shows that *PushdownDB* is on average 30% cheaper and 6.7× faster than a baseline that does not use S3 Select.**

I. Introduction

Clouds offer cheaper and more flexible computing than "on-prem". Not only can one add resources on the fly, the large cloud vendors have major economies of scale relative to "on-prem" deployment. Modern clouds employ an architecture where the computation and storage are disaggregated — the two components are independently managed and connected using a network. Such an architecture allows for independent scaling of computation and storage, which simplifies the management of storage and reduces its cost. A number of data warehousing systems have been built to analyze data on disaggregated cloud storage, including Presto [1], Snowflake [2], Redshift Spectrum [3], among others.

In a disaggregated architecture, the network that connects the computation and storage layers can be a major performance bottleneck. Two intuitive solutions are *caching* and *computation pushdown*. With caching, a compute server loads data from the remote storage and caches it in main memory or local storage, amortizing the network transfer cost. Caching has been implemented in Snowflake [2] and Redshift Spectrum [3], [4]. With computation pushdown, a database management system (DBMS) pushes its functionality as close to storage as possible. Previous research [5] and systems (e.g., Britton-Lee IDM 500 [6], Oracle Exadata server [7], and IBM Netezza machine [8]) have shown that this can significantly improve performance.

Recently, Amazon Web Services (AWS) introduced a feature called "S3 Select", through which limited computation can be pushed onto their shared cloud storage service called S3 [9]. This provides an opportunity to revisit the question of how to divide query processing tasks between S3 storage nodes and normal computation nodes. The question is nontrivial as the limited computational interface of S3 Select allows only certain simple query operators to be pushed into the storage layer, namely selections, projections, and simple aggregations. Other operators require new implementations to take advantage of S3 Select. Moreover, S3 Select pricing can be more expensive than computing on normal EC2 nodes.

In this paper, we set our goal to understand the performance of computation pushdown when running queries in a cloud setting with disaggregated storage. Specifically, we consider filter (with and without indexing), join, group-by, and top-K as candidates. We implement these operators to take advantage of computation pushdown through S3 Select and study their cost and performance. We show dramatic performance improvement and cost reduction, even with the relatively high cost of S3 Select. In addition, we analyze queries from the TPC-H benchmark and show similar benefits of performance and cost. We point out the limitations of the current S3 Select service and provide several suggestions based on the lessons we learned from this project. To the best of our knowledge, this is the *first extensive study of pushdown computing for database operators in a disaggregated architecture.* A more detailed description of this work can be found in [10].

II. Data Management in the Cloud

Cloud providers such as AWS offer a wide variety of computing instances (i.e., EC2: Elastic Compute Cloud) and storage services (i.e., EBS: Elastic Block Store, EFS: Elastic File System, and S3: Simple Storage Service). Compared to other storage services, S3 is a highly available object store that provides virtually infinite storage capacity for regular users with relatively low cost, and is supported by many popular cloud databases, including Presto [1], Hive [11], Spark SQL [12], Redshift Spectrum [3], and Snowflake [2]. The storage nodes in S3 are separate from compute nodes. Hence, a DBMS uses S3 as a storage system and transfers needed data over a network for query processing.

To reduce network traffic and the associated processing on compute nodes, AWS released a new service called *S3 Select* [9] in 2018 to push limited computation to the storage nodes. At the current time, S3 Select supports only selection,

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00174

projection, and aggregation without group-by for tables using the CSV or Parquet [13] format; the storage nodes scan rows in the table and return only qualifying rows to the compute node.

Storage	$0.022/GB/month
Data transfer	free within same region; $0.09/GB out of AWS
S3 Select	scan: $0.002/GB; return: $0.0007/GB
Network request	$0.0004 per 1000 requests
Computation	$2.128 per hour (r4.8xlarge)

TABLE I: S3 query cost breakdown (region `us-east-1`).

The dollar cost of queries is a crucial factor, since it is one of the main reasons to migrate an application from "on-prem" to the cloud. Table I shows the typical value of five cost components when using S3. Since the storage cost does not depend on the frequency of access, we exclude it when calculating query cost in this paper. Servers in our experiments are within the same region as the S3 data. Therefore, we do not pay any data transfer cost. The S3 Select cost is paid based on the amount of data scanned and returned only when S3 Select is used. Network requests cost are charged by the number of HTTP requests; computation cost is charged based on the instance type and how long the virtual machine runs. Data scan and transfer and computation are typically the major components in overall query cost for S3 Select.

A. PushdownDB

In order to explore how a database can leverage S3 Select to improve performance and/or reduce cost, we implemented a bare-bone row-based parallel DBMS testbed, called *PushdownDB*. *PushdownDB* represents a query plan as a directed acyclic graph of operators and executes in a pipelined fashion using multiple Python processes. A few performance optimizations are implemented, including disabling SSL as we expect analytics workloads are typically run in a secure environment and using the Pandas library [14] to represent tuples as data frames. While we could not match the performance of the more mature Presto system on all queries, we obtained competitive performance. The source code of *PushdownDB* is available on GitHub at https://github.com/yxymit/s3filter.git, and is implemented in a mixture of C++ and Python.

Experimental Setup. Experiments in this paper are performed on an r4.8xlarge EC2 instance, which contains 32 physical cores, 244 GB of main memory, and a 10 GigE network. The machine runs Ubuntu 16.04.5 LTS. *PushdownDB* is executed using Python version 2.7.12.

Unless otherwise stated, all experiments use the same 10 GB TPC-H dataset in CSV format. To facilitate parallel processing, each table is partitioned into multiple objects in S3. The techniques discussed in this paper do not make any assumptions about how the data is partitioned.

III. SQL Operators in S3 Select

This section discusses how *PushdownDB* accelerates SQL operators using S3 Select. Specifically, we discuss four operators: filter, join, group by, and top-K.

A. Filter

Both hash indexes and tree-based indexes are widely used in database systems. Neither implementation, however, is a good fit for a cloud storage environment because a single index lookup requires multiple accesses to S3 incurring long network delays. To avoid this problem, we designed an index table that contains the values of the columns to be indexed, as well as the byte offsets of indexed records in that table. Specifically, an index table has the following schema (assuming the index is built on a single column).

```
|value|first_byte_offset|last_byte_offset|
```

Accessing an indexed table comprises two phases. In phase 1, an S3 Select request with the lookup predicate is used to retrieve the byte offsets from the index. In phase 2, the returned byte offsets are used to directly load the corresponding rows from the data table, by sending regular S3 requests for individual rows.

	selectivity=10^{-5}		selectivity=10^{-3}	
	Time	Cost	Time	Cost
Server-side	21.7s	1.3¢	21.3s	1.3¢
S3-side	1.38s	1.5¢	1.82s	1.6¢
Indexing	1.74s	0.4¢	10.7s	3.4¢

TABLE II: Runtime and cost of filter algorithms.

Table II shows the runtime and cost of different filtering algorithms for two selectivities, 10^{-5} and 10^{-3}. Server-side filter loads all the data from S3 into the compute node and performs the filter there. S3-side filter pushes the predicate to S3 using S3 select. S3-side filter is $10\times$ faster than server-side filter with a small increase in cost. S3-side indexing has similar performance as S3-side filter but $4\times$ lower price when the filter is highly selective; the performance of indexing degrades when the filter is less selective due to the cost of S3 requests for individual rows.

B. Join

PushdownDB supports three hash join algorithms: *Baseline Join*, *Filtered Join*, and *Bloom Join*. Baseline join performs the query logic in the compute node without S3 Select. Filtered join pushes down selection and projection using S3 Select to the storage side. In the following we focus on Bloom join. After the hash build phase, a Bloom filter is constructed based on the join keys in the first table and is sent as an S3 Select request to load a filtered version of the second table.

The Bloom filter [15] in *PushdownDB* contains a *bit array* of length m and k different *hash functions*. To add an element, the k hash functions are applied to the element. The output of each hash function is a position in the bit array, which is then set to 1. To query an element, the same k hash functions are applied and the element may be in the set if all the corresponding bits are set. We use universal hashing [16] to implement our hash functions, which can be generalized as:

$$h_{a,b}(x) = ((a \times x + b) \bmod n) \bmod m$$

Where n is a prime $\geq m$. a and b are random integers between 0 and $n-1$, where $a \neq 0$.

In order to push the Bloom filter logic into S3, in *PushdownDB*, we use strings of 1's and 0's to represent the bit array. The following example shows what an S3 Select query containing a Bloom filter would look like.

```
SELECT ...
FROM S3Object
WHERE SUBSTRING('1000011...111101101',
  ((69 * CAST(attr as INT) + 92) % 97) % 68 + 1, 1) = '1'
```

We evaluate the performance of different join algorithms using the following SQL query. We change `upper_bal` to vary selectivity on the `CUSTOMER` table. The false positive rate for the Bloom filter is 0.01.

```
SELECT   SUM(O_TOTALPRICE)
FROM     CUSTOMER, ORDER
WHERE    O_CUSTKEY = C_CUSTKEY AND
         C_ACCTBAL <= upper_bal
```

	upper_bal=-950		upper_bal=-450	
	Time	Cost	Time	Cost
Baseline join	14.5s	0.86¢	14.8s	0.88¢
Filtered join	13.7s	1.3¢	13.9s	1.3¢
Bloom join	2.7s	0.54¢	10.6s	1.03¢

TABLE III: Runtime and cost of join algorithms.

Table III shows the runtime and cost of different join algorithms for two selectivities on the customer table. Baseline and filtered joins perform similarly since they only apply selection to the smaller customer table and load the entire orders table, incurring a similarly large amount of network traffic. Bloom join achieves higher performance and lower cost than either as the high selectivity on the customer table is encapsulated by the Bloom filter, which significantly reduces the number of returned rows for the larger orders table. As the predicate on the customer table becomes less selective, Bloom join's performance degrades as fewer records can be filtered.

C. Group-By

The current S3 Select supports simple aggregation on individual attributes but not with a group-by clause. This section explores *S3-side Group-by* and compares it to *Server-side Group-by* and *Filtered Group-by*. S3-side group-by pushes the group-by logic entirely into S3 and thus minimizes the amount of network traffic. The execution contains two phases.

The first phase runs a projection using S3 Select to load columns in the group-by clause. The compute node then collects the unique values in the grouping columns. In the second phase, *PushdownDB* uses S3 Select to perform aggregation for each individual group that the first phase identified. The following SQL code shows such an example:

```
SELECT sum(CASE WHEN c_nationkey = 0
           THEN c_acctbal ELSE 0 END),
       sum(CASE WHEN c_nationkey = 1
           THEN c_acctbal ELSE 0 END)
       ...
FROM customer;
```

Table IV shows the runtime and cost for different group-by algorithms. We use a synthetic 10 GB table with 20 columns.

	2 groups		8 groups		32 groups	
	Time	Cost	Time	Cost	Time	Cost
Server-side	63.3s	3.7¢	64.5s	3.8¢	65.1s	3.8¢
Filtered	39.6s	4.6¢	39.6s	4.6¢	39.5s	4.6¢
S3-side	9.6s	4.6¢	31.7s	5.9¢	103.3s	10.1¢

TABLE IV: Runtime and cost of group-by algorithms.

The first 10 columns contain group IDs with varying number of groups of uniform size; the other 10 columns contain floating point numbers and are the fields to be aggregated. The performance of server-side and filtered group-by (which pushes projections using S3 Select) does not change with the number of groups, because both algorithms must load all the rows from S3 to the compute node.

S3-side group-by performs $4.1\times$ better than filtered group-by when there are only a few unique groups. Performance degrades, however, when more groups exist. This is due to the increased computation overhead that is performed by the S3 servers. Although the three algorithms have relatively high variation in their runtime numbers, the cost numbers are relatively close until the number of group is large, where the compute cost of S3-side group-by increases significantly.

D. Top-K

Top-K selects the maximum or minimum K records from a table according to a specified expression. In this section, we discuss a sampling-based approach that improves the efficiency of top-K using S3 Select. The algorithm runs in two phases:

The first phase loads a random sample of S ($> K$) records from the table and uses the K^{th} smallest value as the threshold. In the second phase, the algorithm uses S3 Select to load all records below the threshold and uses a heap to select the top K.

Assume each row contains B bytes, the table contains N rows, and only a fraction ($\alpha \leq 1$) of a record is needed during the sampling, which is uniformly random. The total number of bytes loaded from S3 during the two phases are:

$$D_1 = \alpha SB \qquad D_2 = KNB/S$$

The total amount of data loaded from S3 ($D = D_1 + D_2$) achieves its lowest value of $2B\sqrt{\alpha KN}$ when $S = \sqrt{\frac{KN}{\alpha}}$.

	K=1		K=10^2		K=10^4	
	Time	Cost	Time	Cost	Time	Cost
Server-side	44.1	2.61¢	46.0	2.72¢	54.6	3.23¢
S3-side	2.64	1.6¢	3.20	1.65¢	6.62	1.87¢

TABLE V: Runtime and cost of top-K algorithms.

Table V compares the performance of the sampling-based top-K with the baseline that loads the entire table and performs top-K at the server side. K is swept from 1 to 10^4. For the sampling-based algorithm, the sample size is calculated following the analysis above. We observe that both runtime and cost increase as K increases. This is because a larger K requires a bigger heap and also more computation at the server

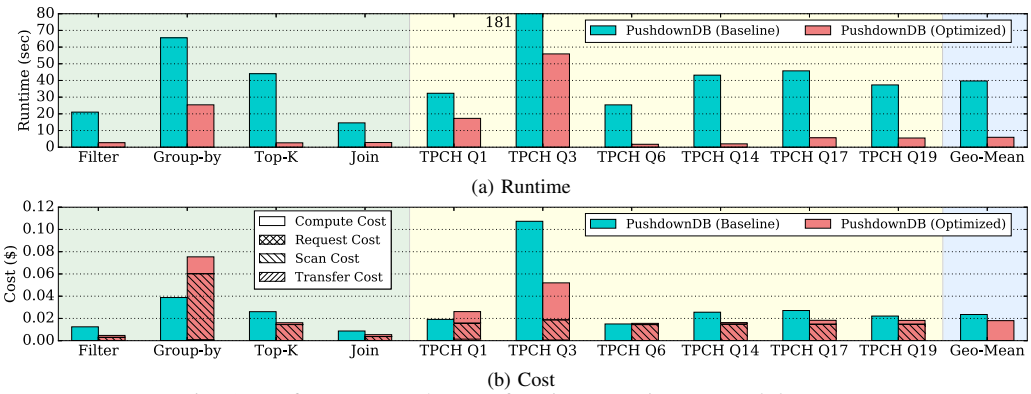

Fig. 1: Performance and cost of various queries on *PushdownDB*.

side. The sampling-based top-K algorithm is consistently faster than the server-side top-K due to the reduction in the amount of data loaded from S3.

E. TPC-H Results

Figure 1 shows the performance and cost of representative queries for each individual operator discussed above, as well as a subset of the TPC-H queries. We compare the performance of server-side execution (baseline) vs. computation pushdown using S3 Select (optimized). The last set of bars shows the geometric mean of all the previous bars. On average, the optimized *PushdownDB* outperforms the baseline *PushdownDB* by 6.7× and reduces the cost by 30%, demonstrating great performance potential of computation pushdown in cloud databases.

IV. Limitations of S3 Select

We have demonstrated substantial performance improvement on common database operators by leveraging S3 Select. In this section, we present a list of limitations of the current S3 Select features and describe our suggestions for improvement.

Suggestion 1: Multiple byte ranges for GET requests. The indexing algorithm discussed in Section III-A sends an HTTP GET request to load each record from the table, causing an excessive number of GET requests and thus performance degradation. Allowing a single GET request to contain multiple byte ranges can mitigate the problem.

Suggestion 2: Index inside S3. A more thorough solution to the problem above is to build the index structure inside S3. This can avoid multiple network messages between S3 and the server which can improve performance.

Suggestion 3: More efficient Bloom filters. Ideally, a Bloom filter should be represented using a bit array for space efficiency. Since the current S3 Select does not support bit-wise operators, *PushdownDB* implements a Bloom filter using a string of 0's and 1's, which is space- and computation-inefficient. We suggest adding bit-wise operators to S3 Select.

Suggestion 4: Partial group-by. In Section III-C, we used the inefficient `CASE` clause to implement S3-side group-by, because group-by is currently not supported in S3 Select. Adding full support of group-by may lead to unbounded memory consumption in the storage node. We suggest adding partial group-by (with limited groups) to S3 to resolve this performance issue.

Suggestion 5: Computation-aware pricing. Across our evaluations on the optimized *PushdownDB*, data scan costs dominate a majority of queries. The current S3 Select charges scanning with a fixed amount regardless of the computation being performed. We believe a fairer pricing model is needed, in which the data scan cost should depend on the workload.

References

[1] "Presto," https://prestodb.io, 2018.

[2] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang *et al.*, "The Snowflake Elastic Data Warehouse," in *SIGMOD*, 2016.

[3] "Amazon Redshift," https://aws.amazon.com/redshift/, 2018.

[4] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon Redshift and the Case for Simpler Data Warehouses," in *SIGMOD*, 2015.

[5] R. B. Hagmann and D. Ferrari, "Performance analysis of several back-end database architectures," *ACM Transactions on Database Systems (TODS)*, vol. 11, no. 1, pp. 1–26, 1986.

[6] M. Ubell, "The Intelligent Database Machine (IDM)," in *Query processing in database systems*. Springer, 1985, pp. 237–247.

[7] R. Weiss, "A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server," *Oracle White Paper. Oracle Corporation, Redwood Shores*, 2012.

[8] P. Francisco, "The Netezza Data Appliance Architecture," 2011.

[9] R. Hunt, "S3 Select and Glacier Select – Retrieving Subsets of Objects," https://aws.amazon.com/blogs/aws/s3-glacier-select/, 2018.

[10] X. Yu, M. Youill, M. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker, "PushdownDB: Accelerating a DBMS using S3 Computation," *arXiv preprint arXiv:2002.05837*, 2020.

[11] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive — A Petabyte Scale Data Warehouse Using Hadoop," in *ICDE*, 2010.

[12] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational Data Processing in Spark," in *SIGMOD*, 2015.

[13] "Apache Parquet," https://parquet.apache.org, 2016.

[14] W. McKinney, "pandas: a Foundational Python Library for Data Analysis and Statistics," *Python for High Performance and Scientific Computing*, pp. 1–9, 2011.

[15] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[16] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.