Optimizing Cloud Data Lakes Queries

Grisha Weintraub

Supervised by Prof. Ehud Gudes and Prof. Shlomi Dolev
Ben-Gurion University of the Negev, Computer Science Department, Beer-Sheva, Israel

Abstract
Cloud data lakes emerge as an inexpensive solution for storing very large amounts of data. The main idea is the separation of compute and storage layers. Thus, cheap cloud storage is used for storing the data, while compute engines are used for running analytics on this data in "on-demand" mode. However, to perform any computation on the data in this architecture, the data should be moved from the storage layer to the compute layer over the network for each calculation. Obviously, that hurts calculation performance and requires huge network bandwidth. Our research focuses on three related topics: (1) identify the key challenges to improving query performance in cloud data lakes, (2) provide a theoretical model that formally defines the problem of poor query performance in cloud data lakes, (3) design a practical solution to the problem and demonstrate its efficiency via large-scale experimental evaluation.

Keywords
data lakes, cloud storage, query optimization

1. Introduction

Traditionally, storage systems have been favoring data locality (meaning they wanted to be as close to the data as possible to speed up calculations on the data). In single-node databases, data locality occurs trivially, whereas in shared-nothing distributed systems, data locality is achieved by performing computation on the same machines that store the data. However, with the rise of cloud technologies, a new family of storage systems has emerged – cloud object stores (e.g., AWS S3 and Azure Blob Storage). These systems provide object storage service through a web interface. Users create buckets, and each bucket may contain multiple binary objects uniquely identified within the bucket by a string key.

Cloud object stores are widely considered to be the most cost-effective storage systems in the world right now [1, 2, 3, 4], and as a result, they are heavily used as the main building block of enterprise data repositories that have come to be known as cloud data lakes [1, 2, 3, 4, 5]. The main feature of the cloud data lakes is that they store data in cloud object stores and as a result do not follow the traditional shared-nothing architecture but, instead, disaggregate compute layer from the storage layer. This new approach is commonly called a *data lake architecture* [5].

VLDB 2023 PhD Workshop, co-located with the 49th International Conference on Very Large Data Bases (VLDB 2023), August 28, 2023, Vancouver, Canada
This research was partially supported by the Israeli Council for Higher Education (CHE) via Data Science Research Center, the Israel Data Science Initiative (IDSI), and the Lynne and William Frankel Center for Computer Science

grishaw@post.bgu.ac.il (G. Weintraub)
0000-0003-4823-4757 (G. Weintraub)

A data lake architecture introduces both benefits and challenges and, not surprisingly, is identified as a promising research direction by recent studies [2, 5]. Our research focuses on the problem of poor query performance in cloud data lakes, and our contributions so far can be summarized as follows:

- We identify the key challenges to improving query performance in cloud data lakes and discuss why existing solutions are not sufficient.
- We provide a theoretical model that formally defines the problem of poor query performance in cloud data lakes. The main component of the model is an optimization problem that clearly defines the relevant trade-offs. We prove that the problem is NP-hard and suggest heuristic algorithms to overcome its hardness.
- We outline future research directions that can significantly improve query performance in cloud data lakes.

The rest of the paper is structured as follows:

- In section 2 we formally define the problem.
- In section 3 we review related work.
- In section 4 we present our preliminary results.
- We conclude in Section 5.

2. Problem Statement

Let us introduce the problem via a simple example first. Consider a typical metric data presented in Table 1. Let us assume that this table is stored in the cloud data lake such that records 1-3 are stored in "file201", records 4-6 are stored in "file170", and records 7-9 in "file051". Files'

Table 1
Sample metric data

	date	metric	val	...	
1	2020-02-10	cpu	47	...	
2	2020-02-14	cpu	58	...	file201
3	2020-02-18	memory	11	...	
4	2020-02-16	memory	8	...	
5	2020-02-20	cpu	88	...	file170
6	2020-02-21	cpu	66	...	
7	2020-03-13	memory	6	...	
8	2020-03-22	cpu	92	...	file051
9	2020-03-28	cpu	71	...	
...	...	...	...	...	

format can be any of the standard supported formats (e.g., Parquet or ORC).

Let us briefly review how the state-of-the-art query engines [6, 7, 8] would execute a typical query on Table 1 stored in the cloud data lake. For example, when a query engine receives a query with ”where” condition ”metric = 'memory' and val > 10”, it reads the files from the storage, scans them in-memory to find the records satisfying the predicate, and returns the result. In our example, only record *3* from *file201* will be returned. However, all the data lake files should be read from the storage and processed; and since production data lakes might contain billions of files [9], this approach is extremely wasteful. So, intuitively, we would like to read only the “relevant” files for a given query (*file201* in our example) and skip all the rest. Let us define the problem formally now.

2.1. Formal problem definition

We model data lake tables according to the standard relational model. Given a set of m domains $D = \{D_1, D_2, \ldots, D_m\}$, a table T is defined as a subset of the Cartesian product $D_1 \times D_2 \times \ldots \times D_m$. Each domain $D_i \in D$ has an associated column name $c_i \in L$, where L is the set of all column names. T is a set of tuples $\{t_1, t_2, \ldots, t_{|T|}\}$, where each tuple t is a set of pairs $\{(c_i : v) \mid c_i \in L, v \in D_i, i \in \{1, \ldots, m\}\}$. T is stored in a cloud object store as a collection of files denoted by $F = \{f_1, f_2, \ldots, f_{|F|}\}$. F is a partition of T, meaning that $\forall_{i \neq j} : f_i \subseteq T, \bigcup F = T, f_i \cap f_j = \emptyset$.

Definition 1 (data lake query). *We define a data lake query Q as a standard SQL query on table T and denote the predicate in the where clause of Q as P_Q. We assume that P_Q is given in a conjunctive normal form (CNF). If tuple t from the file $f \in F$ satisfies P_Q we denote it by $S(P_Q, t)$.*

Definition 2 (query coverage). *Given a data lake query Q, $X \subseteq F$ covers $Q \leftrightarrow \forall_f \in F \setminus X, \neg\exists t \in f, S(P_Q, t)$.*

When X satisfies Definition 2 for some data lake query Q, we say that X *covers* Q (meaning that X contains all the files needed to satisfy Q) and denote it by $Cov(X, Q)$. We call such X a *coverage set* of Q. Note that for any Q, holds $Cov(F, Q)$.

Definition 3 (query tight coverage). *Given a data lake query Q, $X \subseteq F$ tightly covers $Q \leftrightarrow Cov(X, Q) \bigwedge \neg\exists f \in X, \forall t \in f, \neg S(P_Q, t)$*

When X satisfies Definition 3 for some data lake query Q, we say that X tightly covers Q (meaning that X contains all the files needed to satisfy Q and *only* them) and denote it by $TCov(X, Q)$. We call such X a *tight coverage set* of Q and denote it by $TC(Q)$.

If for any data lake query Q, we could (efficiently) compute X such that $TCov(X, Q)$, we would be able to significantly improve query performance in a cloud data lake architecture by accessing only files in X instead of all those in F (and in most real-world scenarios $|X| << |F|$). In fact, as we show below, in many cases finding the exact tight coverage might be too complicated, and we can be content with some coverage set that is not tight but still can help us improve query performance. For such scenarios, the definition of *tightness degree* might be useful.

Definition 4 (tightness degree). *Given a data lake query Q, for any coverage set X of Q, the tightness degree of X is defined as:*

$$TD(X, Q) = \left\{ \begin{array}{ll} 1 - \frac{|X| - |TC(Q)|}{|F| - |TC(Q)|} & \texttt{if } |TC(Q)| < |F| \\ 0 & \texttt{otherwise} \end{array} \right\}$$

Intuitively, the tightness degree shows to what extent the given coverage set is close to the tight coverage set (1 means perfectly close).

Based on the above semantics we can formulate our main research question as follows:

- Can we develop an algorithm, that for any data lake query Q, can find a coverage set of Q, X, such that:
  - tightness degree of X is maximized
  - cost[1] of computing X is minimized
  - as a result of the above, the total execution time of Q is reduced as much as possible

3. Related Work

The most trivial approach for query execution in cloud data lakes is to read all the files. Clearly, F *covers* any Q, but the coverage is far from being *tight* and hence query performance is poor.

One of the first suggested optimizations was data partitioning [6] which is supported by all modern query

[1]cost is measured by the number of files read from the cloud

engines. Unfortunately, only a limited subset of table columns can be used in partitioning, while production tables may contain tens of thousands of columns [10].

Another well-known approach [11] is to attach metadata to each data lake file and use it during the reads to skip irrelevant files. Columnar formats support metadata-based skipping out-of-the-box by storing the metadata and the data in the same file and relying on the fact that cloud object stores support reading of the particular sections of the file. Unfortunately, metadata-based skipping is very sensitive to data distribution and helps only in cases where the data is nicely clustered.

Some cloud providers in some cases (e.g., AWS S3-Select) support pushing the query predicate to the storage layer, so irrelevant records might be filtered out during the read operation and only the relevant records would be returned to the compute layer. This technique is a great optimization, as we do not need to move huge amounts of data between storage and compute layers. However, we still perform a lot of costly filtering operations; the only difference is that the operation is performed in a different place (i.e., in the worst-case scenario all $|F|$ files should be accessed).

Table format (e.g., Delta Lake, Apache Iceberg) [9] is a novel approach to add missing capabilities (e.g., transactions, schema evolution, query optimization) to the data lake architecture. The main idea is to add an additional layer of metadata between the files in the storage and the compute layer. This metadata then can be used, for example, for storing information about schema changes, data mutation, and various statistics to improve query performance. While table format is an important step towards query optimization in cloud data lakes, so far it does not solve the main problem considered in our study: reading of (a lot of) irrelevant files from the storage.

Indexing is the primary method for improving query performance in relational databases but it is rarely used in big data environment. Our preliminary work in [12] presents an indexing scheme for relational data in cloud data lakes where indexes are stored inside the lake and their creation is performed by parallel algorithms. This scheme, however, limits the query model to simple selection/projection queries with a single-column predicate.

4. Our Approach

Finding the tight coverage set of a query might be too costly (in can be proved that in worst-case scenario it cannot be better than $\Omega(F)$). The main idea of our approach is, instead of looking for the *tight* coverage set of the given query, to find *some* coverage set that will result in an optimal *total* query execution time (i.e., we are looking for the coverage set X such that the sum of the cost of finding X and its size is minimized). We focus on the "where" condition of the query and look at each predicate clause separately. There may be many coverage sets associated with each clause, and there may be many ways to compute each of these coverage sets. We assume that for each possible *coverage execution plan* we can estimate (e.g., via statistics, caching, ML models, etc.) what is the expected cost and expected result of each plan. An important observation is that intersection of coverage sets of any subset of the query clauses is a coverage set of the original query. Based on this observation, our optimization scheme consists of the following two steps:

1. Given a query and its estimated values, we find a subset of clauses (and their corresponding coverage plans), such that the sum of their estimated costs and the size of the intersection of the coverage sets is minimized.
2. Then, we execute each of the coverage plans, intersect their results and execute the original query on the files in the intersection only.

Step (1) can be formulated as an optimization problem and we prove that it is NP-hard (by reduction from the set covering problem); we suggest heuristic algorithms to deal with the hardness of the problem. Our result estimation scheme is based on the regular relational databases *selection size estimation* [13], in which we can estimate the number of records satisfying the predicate based on the statistical information. Once we have the estimated number of records (R) for the given clause, we can estimate the corresponding number of files as the number of non-empty bins after randomly throwing R balls into $|F|$ bins. Execution of coverage plans (step 2) is based on our previous work on indexing in cloud data lakes [12, 14].

We built a prototype of our solution; the implementation is available online [2]. We used Apache Spark as the main compute engine and AWS as the cloud provider. For the benchmark, we used the TPC-H dataset [3] with a scale factor of 1 TB. We executed different queries with and without our scheme and achieved up to x19 query performance improvement.

5. Conclusions and Future Research Directions

Data volumes keep growing, and new technologies are being developed all the time to adjust to the constant increase in data traffic. Cloud data lakes are one of the most successful solutions to the big data challenge. They provide great scalability, usability, and cost-efficiency, but perform poorly with interactive queries.

[2]https://github.com/grishaw/data-lake-coverage
[3]https://www.tpc.org/tpch/

We analyze the problem of poor performance of interactive queries in cloud data lakes and identify the main obstacles achieving better performance. We formally define the problem by introducing a new concept of the query (tight) coverage set. We use our terminology to define an optimization problem that finds the best coverage set for the given query. We show that the problem is NP-hard and deal with its hardness by suggesting heuristic algorithms. Our solution is based on ideas from relational databases related to indexing and statistics management adjusted to a cloud data lake architecture. The main idea is that the storage resources are usually much cheaper than the compute resources [4, 12], so if we could provably improve query performance by adding more storage it probably would be very useful for many data lake users.

For future research, we are planning to focus on the following directions:

- We can try improving our estimations scheme by training ML models to "guess" expected result of a given query coverage plan.
- We want to extend our basic query model to more complex types (e.g., joins and group by).
- An interesting caching technique can be based on the tight coverage sets: $TCov$ function defines an equivalence relation on a set of all possible queries over the table. So, if for any query Q we could tell for each equivalence class it belongs, we would be able to cache tight coverage set per equivalence class on the query engine side and by that improve query performance even more while using limited memory resources.
- We want to apply our optimization techniques in real-world applications (we already have initial results in a multidisciplinary project where we analyze large-scale genomic data with cloud data lakes[15, 16]).

References

[1] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, et al., Delta lake: high-performance acid table storage over cloud object stores, Proceedings of the VLDB Endowment 13 (2020) 3411–3424.

[2] M. Armbrust, A. Ghodsi, R. Xin, M. Zaharia, Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics, in: Proceedings of CIDR, 2021.

[3] Y. Yang, M. Youill, M. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, M. Stonebraker, Flexpushdowndb: Hybrid pushdown and caching in a cloud dbms, Proceedings of the VLDB Endowment 14 (2021) 2101–2113.

[4] J. Tan, T. Ghanem, M. Perron, X. Yu, M. Stonebraker, D. DeWitt, M. Serafini, A. Aboulnaga, T. Kraska, Choosing a cloud dbms: architectures and tradeoffs, Proceedings of the VLDB Endowment 12 (2019) 2170–2182.

[5] D. Abadi, A. Ailamaki, D. Andersen, P. Bailis, M. Balazinska, P. A. Bernstein, P. Boncz, S. Chaudhuri, A. Cheung, A. Doan, et al., The seattle report on database research, Communications of the ACM 65 (2022) 72–79.

[6] J. Camacho-Rodríguez, A. Chauhan, A. Gates, E. Koifman, O. O'Malley, V. Garg, et al., Apache hive: From mapreduce to enterprise-grade big data warehousing, in: Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 1773–1786.

[7] 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: Proceedings of the 2015 ACM SIGMOD international conference on management of data, 2015, pp. 1383–1394.

[8] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, et al., Presto: Sql on everything, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), IEEE, 2019, pp. 1802–1813.

[9] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, M. Zaharia, Analyzing and comparing lakehouse storage systems, in: Proceedings of CIDR, 2023.

[10] P. Edara, M. Pasumansky, Big metadata: when metadata is big data, Proceedings of the VLDB Endowment 14 (2021) 3083–3095.

[11] G. Moerkotte, Small materialized aggregates: A light weight index structure for data warehousing, in: VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, 1998, pp. 476–487.

[12] G. Weintraub, E. Gudes, S. Dolev, Needle in a haystack queries in cloud data lakes., in: EDBT/ICDT Workshops, 2021.

[13] A. Silberschatz, H. F. Korth, S. Sudarshan, et al., Database system concepts, volume 5, McGraw-Hill New York, 2002.

[14] G. Weintraub, E. Gudes, S. Dolev, Indexing cloud data lakes within the lakes, in: Proceedings of the 14th ACM International Conference on Systems and Storage, 2021, pp. 1–1.

[15] G. Weintraub, N. Hadar, E. Gudes, S. Dolev, O. Birk, Analyzing large-scale genomic data with cloud data lakes, in: Proceedings of the 16th ACM International Conference on Systems and Storage, SYSTOR '23, 2023, p. 142.

[16] N. Hadar, G. Weintraub, E. Gudes, S. Dolev, O. S. Birk, Geniepool: genomic database with corresponding annotated samples based on a cloud data lake architecture, Database 2023 (2023) baad043.