

# Needle in a haystack queries in cloud data lakes

Grisha Weintraub  
grishaw@post.bgu.ac.il

Ben-Gurion University of the Negev  
Beer-Sheva, Israel

Ehud Gudes  
ehud@cs.bgu.ac.il

Ben-Gurion University of the Negev  
Beer-Sheva, Israel

Shlomi Dolev  
dolev@cs.bgu.ac.il

Ben-Gurion University of the Negev  
Beer-Sheva, Israel

## ABSTRACT

Cloud data lakes are a modern approach for storing large amounts of data in a convenient and inexpensive way. Query engines (e.g. Hive, Presto, SparkSQL) are used to run SQL queries on data lakes. Their main focus is on analytical queries while random reads are overlooked. In this paper, we present our approach for optimizing *needle in a haystack* queries in cloud data lakes. The main idea is to maintain an index structure that maps indexed column values to their files. According to our analysis and experimental evaluation, our solution imposes a reasonable storage overhead while providing an order of magnitude performance improvement.

## 1 INTRODUCTION

Data lakes are a relatively new concept, which has recently received much attention from both the research community and industry [2, 29, 41]. There is no formal definition of the term *data lake*, but it is commonly described as a centralized repository containing very large amounts of data in their original (or minimally processed) format. The main idea behind this approach is that instead of loading the data into traditional data warehouses, which require scrupulous schema design, and a high development and maintenance cost, the data is simply streamed into a distributed storage system (e.g. HDFS [43]), and from that moment becomes available for analytics. Cloud data lakes are managed by cloud providers and store their data in cloud object stores, as AWS S3 [12], Google Cloud Storage [31], and Azure Blob Storage [37]. For analytics, compute engines (e.g. Spark [47], MapReduce [25]) are used "on-demand". They access cloud data lakes via simple get/put API over the network, and this separation of storage and compute layers results in a lower cost and allows independent scaling of each layer (Fig. 1).

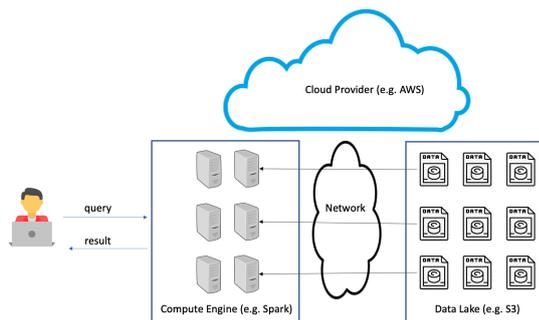


Figure 1: Data Lakes Architecture

A plethora of new systems, often referred to as *query engines* [1], have been developed in recent years to support SQL queries

on top of data lakes. Hive [18], originally built at Facebook, was the first such system. In Hive, SQL-like queries are compiled into a series of map-reduce jobs and are executed on MapReduce [25] framework. A somewhat similar approach is implemented in Spark SQL [13], where SQL queries are running on top of the Spark engine [47].

Another group of query engines do not use general-purpose frameworks such as MapReduce or Spark but rely on their own engines, built on the ideas from parallel databases. The first such system was Google's Dremel [36], which inspired many modern query engines; most well-known are Drill [32], Presto [42] and Impala [33].

The following techniques are commonly associated with the query engines running on data lakes:

- **in-situ processing:** In-situ means that the data is accessed *in-place* (i.e. in HDFS or cloud object store), and there is no need to load the data into the database management system (DBMS). Thanks to this feature, the same data lake can be queried simultaneously by different query engines without any difficulties.
- **parallel processing:** The data is accessed in parallel (usually by a cluster of dedicated machines).
- **columnar storage:** The main idea is that instead of storing the data row by row, as is customary in relational databases, the data is stored column by column. The most important benefits of such column-wise storage are fast projection queries and efficient column-specific compression and encoding techniques. Some of the popular open source formats are ORC and Parquet [28].
- **data partitioning:** The idea is simple and yet powerful. The data is partitioned horizontally based on some of the input columns. Then, files related to each partition are stored in a different directory in the storage system (see an example below).

Let us demonstrate how these techniques look in a typical real-world scenario. Consider a large enterprise company receiving a lot of events (e.g. from IoT sensors) and stores them in a data lake. The data lake would look like a collection of files (Fig. 2) stored in some distributed storage system. In our example, the data lake is partitioned by *year* and *month* columns, so for example events from March 2020 are located in the path *data-lake/year:2020/month:03/*. Events are stored in a columnar format *Parquet*, so queries on specific columns would scan only relevant chunks of data. When a query engine executes a query, it scans all the relevant files in parallel and returns the result to the client.

Data lakes and query engines are optimized for analytic queries and a typical query calculates aggregate values (e.g. sum, max, min) over a specific partition. For such queries, all the files under specified partition should be scanned.

However, sometimes users may be merely interested in finding a specific record in the data lake (a.k.a. needle in a haystack). Consider for example a scenario where a specific event should be fetched from the lake based on its unique identifier for troubleshooting or business flow analysis. Another important use

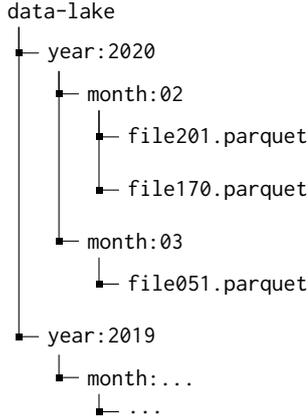


Figure 2: Data lake layout in storage system

case is managing the General Data Protection Regulation (GDPR) compliance [45], where records related to a specific user should be found in the data lake upon user request. Unfortunately, even when the requested information resides in a single file, query engines will perform a scan of all the files in the data lake (can be millions). Obviously, this behavior results in a long query latency and an expensive computation cost.

In this paper, we present our approach for optimizing "needle in a haystack" queries in cloud data lakes. The main idea is to build an index structure that maps indexed column values to the files that contain those values. We assume enormous data volumes in cloud data lakes, and hence, we need our index to be highly scalable in both compute and storage sense. For scalable compute, we utilize parallel processing paradigm [25, 47]; for scalable storage, we design a new storage format that allows unlimited scaling while minimizing the number of network operations during reads.

Our main contributions are as follows:

- Development of a novel index structure that allows speeding up random queries in cloud data lakes.
- A prototype implementation of our solution and its experimental evaluation in a real-world cloud data lake.

The rest of the paper is structured as follows. In Section 2, we formally define the problem. In section 3, we describe the proposed solution. In section 4, we provide a performance analysis of our solution. In Section 5, we introduce our proof-of-concept implementation and provide an experimental evaluation thereof. In Section 6, we review related work. We conclude in Section 7.

## 2 PROBLEM STATEMENT

We model a data lake as a set of tuples  $D = \{ \langle c_1 : v_1, c_2 : v_2, \dots, c_n : v_n \rangle \mid \forall 1 \leq i \leq n, c_i \in C, v_i \in V \}$  where  $C$  is the set of column names and  $V$  is a set of column values (see notation in Table 5). Data lake  $D$  is stored in a distributed storage system as a collection of  $N$  files, denoted by  $F = \{f_1, f_2, \dots, f_N\}$ .

Query engines support SQL queries on data lakes. In this paper our focus is on simple *selection* queries of type:

```
SELECT C1, C2, ...
FROM DATA-LAKE
WHERE SOME-COLUMN = SOME-VALUE
```

Files that contain tuples that satisfy query  $Q$  are denoted by  $F(Q)$ . As already mentioned in the previous section, the current

Table 1: Index content example

Column Name	Column Value	File Names
event-id	207	file051, file033
event-id	350	file002
event-id	418	file170, file034
...	...	...
client-ip	192.168.1.15	file170
client-ip	172.16.254.1	file883, file051

approach to perform query  $Q$  is to scan all  $F$  files in parallel, which is wasteful as only  $F(Q)$  files should be scanned and in many cases  $|F| \gg |F(Q)|$ . If we knew  $F(Q)$  for a given  $Q$  we would read only relevant files and so will be able to significantly improve query performance. Thus the problem we are trying to solve in this paper can be defined as – "construct a function  $getFiles(Q)$  that receives a query  $Q$  and returns  $F(Q)$  - the list of files that contain tuples that satisfy  $Q$ ".

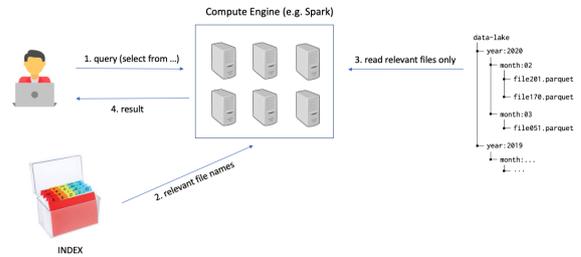


Figure 3: System model

Fig. 3 presents a high-level diagram of the considered system model.

- (1) A client submits a query to the query engine
- (2) A query engine queries the index to get relevant file names
- (3) A query engine reads relevant files from the data lake
- (4) A query engine computes query result and returns it to the client

An obvious approach to implement  $getFiles(Q)$  is to create a simple inverted index [40] that stores a mapping between the values of relevant columns and the files that contain these values (Table 1). An open question is: what is the best way to do that in a big data environment. More formally, we are looking for a solution that will satisfy the following constraints:

- (1) *scalability*: we want our index to be able to handle very large and growing amounts of data
- (2) *performance*: we need our index to be able to respond to the lookup queries in a reasonable time (optimally sub-seconds)
- (3) *cost*: we want to minimize the monetary cost of our solution as much as possible

The trivial solution would be to store our index in a distributed NoSQL database (e.g Cassandra [34], DynamoDB [11]) and update it with parallel compute engines (e.g Spark, MapReduce). However, while this approach may satisfy the first two of our requirements (scalability and performance), its monetary cost is far from being optimal. Let us show why by using some real numbers.

We have two main alternatives for using a database service in the cloud environment: *IaaS* and *DaaS*.

**Table 2: Storage cost comparison**

	cost of 1TB/year	pricing info references
Cassandra on EC2	16,164 \$	[5, 9]
DynamoDB	2,925 \$	[4]
S3	276 \$	[6]

- (1) IaaS: "Infrastructure-as-a-Service" model provides users with the requested number of virtual machines where they can install DBMS and use it as they wish (e.g. Cassandra on top of EC2).
- (2) DaaS: "Database-as-a-Service" model provides users with database API without the need for managing hardware or software (e.g. DynamoDB).

In Table 2<sup>1</sup>, we compare the monetary cost of storing data in each one of these alternatives against storing it in a cloud object store (similarly to how the data is stored in cloud data lakes). It clearly can be seen that an object store is at least one order of magnitude lower-priced than any other option. Since our focus is on real-world cloud data lakes (i.e. petabytes and beyond scale), we need to support indexes of very large volumes as well (index on a unique column of data lake  $D$  will need  $O(|D|)$  storage). Thus, potential monetary savings of implementing indexing by using object stores, instead of the naive solution based on key-value stores, can be tremendous.

This observation is not new, and object stores superiority in cost is a well-known fact in both industry [24] and research community [44]. However, object stores are not suitable for every case. More specifically, when comparing them to key-value stores, they have the following caveats:

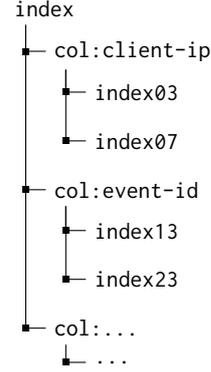
- (1) higher latency : the actual numbers may vary, but usually object stores have higher latency than DBMS [44]
- (2) limited requests rate : while requests rate is nearly unlimited in scalable databases, in object stores there is an upper limit (typically around thousands per second [38])
- (3) optimized for large objects : object stores are optimized for storing large objects (typically GB's [8]) and hence are not suitable for systems that need random reads
- (4) no random writes : the only way to mutate objects in object stores is to replace them, which may be problematic for systems that need random writes

While the first two points ("higher latency" and "limited requests rate") are usually acceptable in OLAP systems, the last two ("large objects" and "no random writes") introduce significant implementation challenges. In our solution, we address both these challenges.

To summarize, in this work, our research goal is twofold:

- to implement  $getFiles(Q)$  and thereby improving random reads performance in cloud data lakes
- to build our solution on top of object stores so the additional monetary cost imposed by our scheme will be as low as possible

<sup>1</sup>For simplicity, we ignore some factors that cannot change the general trend, like the maintenance cost of Cassandra, the cost of requests in S3 and DynamoDB, and the fact the data in object stores can be compressed. Our goal is only to show that object stores are significantly cheaper than the other options. We also provide numbers for specific systems (Cassandra, DynamoDB, AWS), but the same cost trend is preserved in other similar systems as well.



**Figure 4: Index storage layout example**

### 3 OUR APPROACH

In sections 3.1, 3.2, and 3.3 below, we explain how exactly we store, compute, and use our index. In section 3.4, we discuss potential optimizations to our basic scheme.

#### 3.1 Index Storage

We store our index as a collection of files partitioned by an indexed column name (Fig. 4). Each index file is responsible for some range of column values and stores relevant information about this range in a format depicted in Table 3.

There are two sections in the index file - *data* and *metadata*. The data part consists of a map between a column value and a list of corresponding file names ordered by a column value. The map is divided into  $K$  chunks based on the predefined number of values per chunk  $M$ , such that each of the  $K$  chunks contains a mapping for the  $M$  consecutive values of the particular indexed column. For example, in the index presented in Table 3, we have  $M = 3$ , and the first chunk contains a mapping between 3 column values (207, 350, 418) and the file names that contain tuples with these values.

Metadata stores information about the location of data chunks inside the index file along with their "border" values. Considering the example in Table 3 again, from the metadata section we learn that the firsts chunk's minimal value is 207, the maximal value is 418, and its location in the file is in offsets 0:2048.

For very big data lakes, we will have many index files and then will start experiencing the same problem we are trying to solve in data lakes (many irrelevant files are accessed). We solve this issue by managing an index of indexes (i.e. root index) - a single file that stores a mapping between the index file name and its border values (Table 4). In Section 3.3 below, we will explain how the root index and metadata sections are used by the index users.

#### 3.2 Index Computation

Since data lakes store enormous volumes of data, our index will be very large as well. To be able to compute such an index in a reasonable time we would like to be able to compute it in parallel by using common parallel compute engines like MapReduce [25] and Spark [47].

Fortunately, we can easily do that by a simple map-reduce flow. Pseudo-code (in Spark-like syntax) of the algorithm for creating an index on a specific column in a given data lake is presented in Algorithm 1.

**Table 3: Index file example**

Data	207	file051, file033
	350	file002
	418	file170, file034
	513	file0002
	680	file443, file001, file033
	799	file883
...	...	
Metadata	1	min:207 max:418 offset:0
	2	min:513 max:799 offset:2048
	...	...

**Table 4: Root index example**

Column Name	Min	Max	Index File Name
event-id	207	560	index13
event-id	590	897	index23
...	...	...	...

In the map phase, each worker emits  $(col\text{-}value, file\text{-}name)$  pairs and in reduce phase map results are grouped into  $(col\text{-}value, List<file\text{-}name>)$  map. Then, the result is written into the distributed storage in our custom format (presented in Table 3 and denoted in the pseudo-code as *my.index*). Finally, we create the *root index* file by scanning metadata sections of all the created index files and taking minimum and maximum values from each file.

Algorithm 1 creates an index for an existing data lake (i.e. static scenario). In a dynamic scenario, bulks of new files are being added to the lake periodically during scheduled Extract-Transform-Load (ETL) processes. To support a dynamic scenario, we should be able to update our index in accordance with the new data.

Our algorithm for index update, presented in Spark-like syntax in Algorithm 2, runs in parallel and tries to modify as few existing index files as possible. It works like that:

- First (line 2), we read the root index to get "ranges" - a set of tuples  $(min, max, index\text{-}file\text{-}name)$  for a given column (the result may look similar to Table 4).
- Then (lines 3-4), we use broadcast join technique [16] to compute "labeled-delta" - values from delta files labeled with the corresponding "index-file-name" from "ranges" set. Considering the example in Tables 1-4, the result would look as a set of triples  $(event\text{-}id, data\text{-}file\text{-}name, index\text{-}file\text{-}name)$ ; for example,  $(450, file1111, index13)$ ,  $(645, file1112, index23)$ .
- Finally (lines 5-8), we use index file names computed in labeled-delta, to identify index files that should be changed. We read these files and combine their data with the delta files. Then, we apply Algorithm 1 on this data and override relevant existing index files and the root index.
- Note that we need to re-partition our data by "index-file-name" (line 7) to ensure that there will not be a mix between different index files and each index file will be computed by a different process.

### 3.3 Index Usage

In this section we explain how to use our index format to implement *getFiles(Q)* (pseudo-code is presented in Algorithm 3).

Upon receiving a "needle in a haystack" query  $Q$  and a path to index location, we first extract the column name and the column value from the "where" clause of the query (line 2). We then read the root index and find to which index file the given value belongs (lines 3-4). Once we know what index file has information about our value, we perform the following:

- we read the metadata part of the file
- we find to which data chunk our value belongs based on the min/max values.
- once found to which chunk our value belongs, we read the  $(column\text{-}value, List<file\text{-}name>)$  map from the specified offset
- if our value appears in the map we are done and the method returns the file names list
- otherwise we know that requested value is not in the index so we exit with a relevant flag

Note that Algorithm 3 does not require parallel processing and can run as a standalone function.

**Algorithm 1 Create Index**

```

1: procedure CREATEINDEX(datalake-path, index-path, col-name)
2:   datalake = read(datalake-path)
3:   index = datalake.groupBy(col-name).agg(collect("f-name"))
4:   index-sorted = index.orderBy(col-name)
5:   index-sorted.write.format("my.index").save(index-path)
6:   create-root-index(index-path)
7: end procedure

```

**Algorithm 2 Update Index**

```

1: procedure UPDATEINDEX(path-to-new-files, index-path, col-name)
2:   ranges = getIndexRanges(index-path, col-name)
3:   delta = read(path-to-new-files).select(col-name, "f-name")
4:   labeled-delta = delta.join(broadcast(ranges))
5:   old-index = read(labeled-delta.select(index-file-name))
6:   new-index = labeled-delta.union(old-index)
7:   new-index.repartition(index-file-name)
8:   run CreateIndex on new-index and override old-index files and
   the root-index
9: end procedure

```

**Algorithm 3 Get Files**

```

1: procedure GETFILES(Q, index-path)
2:   col-name, col-value = extract-col-values(Q)
3:   root-index = readRootIndex(index-path)
4:   index-file = root-index.getFile(col-name, col-value)
5:   meta = readMetadata(index-file)
6:   chunk-id = meta.getChunkIdByValue(col-value)
7:   map = readDataChunk(index-file, chunk-id)
8:   fileNames = map.get(col-value)
9:   if fileNames is not empty then
10:     return fileNames
11:   else
12:     return VALUE-NOT-EXISTS
13:   end if
14: end procedure

```

### 3.4 Optimizations

In this section, we briefly discuss several potential optimizations that can be done to our basic scheme. We plan to deepen some of them in our future work.

- **Bloom filter:** In Algorithm 3, we perform two reads from the index file that may contain our queried value even if the value does not exist in the data lake at all. A possible solution to this issue is adding a Bloom filter [17] containing all the column values in the index file to the metadata section and check it before querying the index data chunk. This way, in most cases, we will perform a single read operation for non-existing value but will have a larger metadata section in each index file.
- **Caching:** By caching the root index and metadata sections on the query engine side, we can significantly reduce the number of network operations per query.
- **Compression/Encoding:** We can apply different encoding and compression techniques to reduce the storage overhead of our index. For example, instead of storing file names lists as strings, we can use compressed bit vectors [21]; metadata offsets can be encoded with delta encoding [35].

## 4 PERFORMANCE ANALYSIS

In this section, we analyze our solution from a performance perspective. We first show how our index improves queries performance and then we show what is the performance overhead imposed by it.

A cost of query  $Q$  is dominated by a cost of read operations from remote storage. When columnar format is used, irrelevant columns can be skipped, but at least some data should be read from *each* file. Thus, by using a notation from Table 5, we can define a cost of query  $Q$  as:

$$C_Q = C_R \cdot |F| \quad (1)$$

When using our index, we read only relevant data files  $F(Q)$  and perform at most three read operations from the index files (for the root index, metadata, and data chunk). Hence, the cost is reduced to:

$$C_Q = C_R \cdot (|F(Q)| + 3) \quad (2)$$

Thus, the performance is improved by:

$$\frac{|F|}{|F(Q)| + 3} \quad (3)$$

Since our focus is on "needle in a haystack" queries, we assume that  $|F(Q)|$  is a small number (typically below 100), while  $|F|$  in real-world data lakes can reach millions [42]. Thus, we can expect a reduction of query cost by order of magnitude.

Our solution imposes overhead in the following two dimensions:

- **Storage:** Additional storage required for our index has an upper limit of  $|D|$  (in case that all columns are indexed). In a more realistic scenario, when only a few of the data lake columns are indexed, storage overhead is expected to be much lower than  $|D|$ . For example, in our experimental evaluation an index on unique column requires around 5% of the data lake size (exact numbers are presented in Section 5.1)
- **CreateIndex/UpdateIndex:** Algorithms 1 and 2 are supposed to run offline without affecting query performance. Their cost is dominated by the cost of reading relevant

Table 5: Notation

Symbol	Description
$D$	A data lake
$C$	Data lake column names
$V$	Data lake column values
$F$	Data lake files
$N$	Number of files in the data lake
$K$	Number of data chunks in index file
$M$	Number of values in a data chunk
$Q$	A query
$R$	Read operation from remote storage
$F(Q)$	Files that contain tuples that satisfy $Q$
$ x $	Size of object $x$
$C_x$	Cost of operation $x$

data lake files, creating an index by map-reduce flow, and writing the index into a distributed storage.

In summary, our solution imposes reasonable storage and (offline) computation overhead but improves query performance by order of magnitude (equations 1-3).

## 5 IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented a prototype of our solution (Algorithms 1, 3); the implementation and extended results are available on-line [46]. For experimental evaluation, we cooperated with a large enterprise company that manages cloud data lakes at petabyte scale. We used an anonymized subset of the real data lake for our experiments.

The data lake we used is stored in AWS S3 cloud storage. The data lake properties are as follows:

- 1,242 files in Parquet format (compressed with snappy)
- 605,539,843 total records
- each record has a unique identifier (8 bytes)
- 233 columns (mainly strings but also numbers and dates)
- 54.2 GB (total compressed size)

Our evaluation focused on the following stages:

- (1) **Index Creation:** We created an index according to our approach (Algorithm 1) and according to a naive approach based on a key-value store. Then, we compared index creation time and its storage size in each of the approaches.
- (2) **Index Usage:** We queried random values from the data lake in different ways (with and without index) and compared query execution time in different modes.

### 5.1 Index Creation Evaluation

We built an index on two data lake columns:

- **record-id (Long):** unique identifier of each record in the data lake (distinct count - 605,539,843)
- **event-id (String) :** unique identifier of each event in the data lake (distinct count - 99,765,289), each event-id is mapped to 6 record-ids on average.

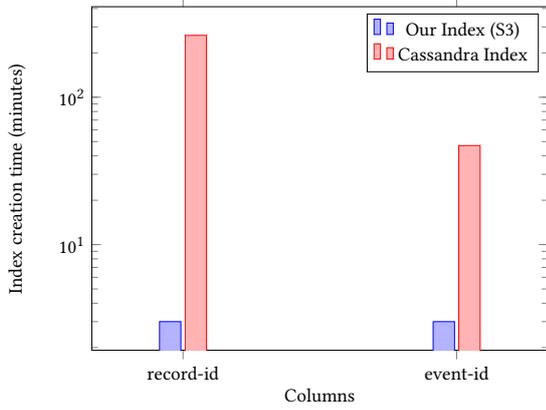
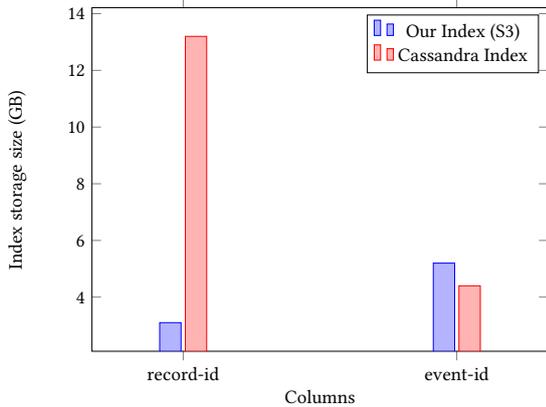
For index creation, we used Apache Spark (2.4.4) running on AWS EMR (5.29.0). EMR cluster had 15 nodes of type r5d.2xlarge (8 cores, 64GB memory, 300GB SSD storage). Our index was stored in AWS S3 as a collection of Parquet files compressed with Snappy and with a chunk size of 10 MB; root-index was stored in a single line-delimited JSON file. The key-value index was stored

**Table 6: Index creation results**

column name	our index (S3)	Cassandra index
record-id (pk)	<u>compute time:</u> 3 min  <u>storage size:</u> 1 root file - 5.5 KB 30 index files - 3.1 GB	<u>compute time:</u> 4 hours, 23 minutes  <u>storage size:</u> 13.2 GB
event-id	<u>compute time:</u> 3 min  <u>storage size:</u> 1 root file - 5.3 KB 30 index files - 5.2 GB	<u>compute time:</u> 47 min  <u>storage size:</u> 4.4 GB

in Apache Cassandra (3.4.4) running on AWS EC2 (3 nodes of type i3.xlarge, a replication factor of 1, LZ4 compression).

Experimental results of index creation are presented in Table 6 and Figures 5, 6.

**Figure 5: Index Creation Time****Figure 6: Index Creation Storage**

It is worth noting, that even though our primary reason to implement the data lake index on top of object stores was motivated by reducing the monetary cost, our experiments provide an additional benefit of this decision. Index creation time of our

approach is an order of magnitude faster than those based on a key-value store. In addition, in our approach, the storage size of the index is much smaller for a unique column and more or less the same for a non-unique column.

## 5.2 Index Usage Evaluation

For index usage evaluation, we performed the following test for the indexed columns (*record-id*, *event-id*):

- (1) get 10 random column values from the data lake
- (2) for each *column-value*, do
  - (a) run "select \* from data-lake where column = *column-value*"
  - (b) print query execution time

We executed this test in the following modes and compared their results.

- (1) SparkSQL with our index (Algorithm 3)
- (2) SparkSQL with Cassandra index
- (3) SparkSQL without index
- (4) AWS Athena [3] without index

SparkSQL was executed on the same cluster as in the previous section (AWS EMR with 15 nodes). AWS Athena (Amazon serverless query engine based on Presto [42]) ran on AWS Glue [7] table created on top of the data lake Parquet files. The average results of the 10 runs for each one of the modes are presented in Fig. 7.

Experimental results confirm our performance analysis in Section 4, as it clearly can be seen that our solution (as well as Cassandra-based) outperforms existing query engines (Spark and Athena) by order of magnitude.

To better understand the trade-offs between our solution and Cassandra-based, we also compared their execution time of calculating *getFiles(Q)*. The results are presented in Fig. 8. We evaluated two versions of our approach, with and without caching of the root index (see Section 3.4). Cassandra latency is lower than that of our approach, but when using a cache, the difference is not significant (less than a second).

To summarize, as expected, adding indexing to data lakes drastically improves queries performance. In this evaluation, we have demonstrated that index can be implemented in two different ways - a trivial approach based on key-value stores and our approach based on cloud object stores. Both approaches improve queries' execution time in an almost identical way (the difference is in milliseconds resolution). However, our approach has the following advantages over the naive solution:

- (1) order of magnitude monetary cost improvement (Table 2)
- (2) order of magnitude index creation time improvement (Fig. 5)
- (3) significantly less storage overhead in some cases (Fig. 6)

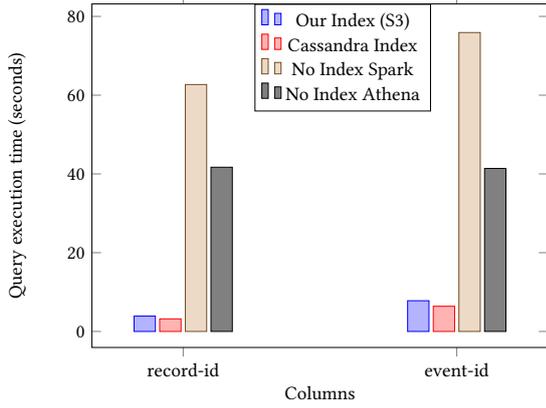


Figure 7: Data Lake Queries Latency

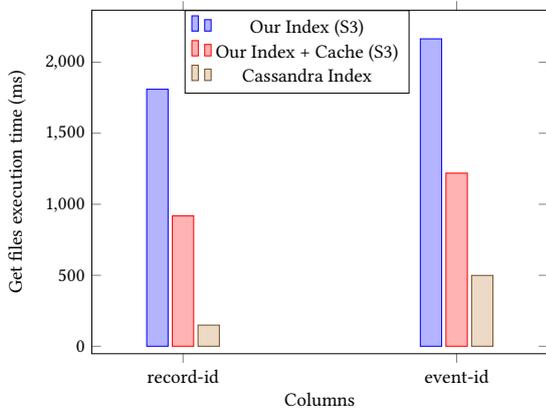


Figure 8: Get Files Latency

## 6 RELATED WORK

Data lakes are a relatively new approach for storing and analyzing large amounts of data. As a new technology, it still lacks a formal definition, and in existing literature, different authors assign different (and sometimes even contradictory) meanings to this concept [29, 41]. In this paper, we stick to the definition from the recent Seattle Report on Database Research [2], where the main characteristic of the data lakes architecture is that it "disaggregates compute and storage, so they can scale independently."

There are several research challenges in the data lakes domain. The most fundamental ones are knowledge discovery [30] and various optimizations [2]. We focus on the latter and study optimization of a specific scenario in data lake systems - "needle in a haystack" queries. Our solution can be seen as a combination of different techniques from databases and big data domains. Thus, we use partitions as in [18] to organize our index according to columns. For scalability, we use root-index as in B+ trees [40] and some distributed databases (e.g. [22]). The format of our index files is similar to columnar formats [28] with multiple chunks of data and a single metadata section. We use Bloom filters [17] to improve the performance of queries on non-existing values. Parallel processing paradigm [25, 47] is used for index creation and updating.

At a high level, our solution resembles a well-known *inverted index* structure which is extensively used in full-text search engines [14, 20, 40, 48]. Inverted index maps words to the lists of documents that contain them, and for data sets of the moderate size, it can be implemented with the standard index structures as a B+ tree or a hash index [40]. For "big data" volumes, distributed inverted indexes were proposed [48]. Two main techniques used in distributed inverted indexing are document-based [14] and term-based [20]. Both these techniques assume combined compute and storage layers, while in our system model, the main assumption is that compute and storage are separated.

In [27], the authors study inverted indexes in *dataspaces* ("collections of heterogeneous and partially unstructured data"). Their main focus is on the heterogeneity of data and very large indexes are not considered (index implementation is based on a single-node Lucene engine [15]); in our solution, the focus is on structured data, yet we expect very large volumes of both data and indexes. In [26], indexing in big data is achieved by enhancing the MapReduce load process via user-defined functions (UDF). However, this solution is limited to a particular system (Hadoop), where compute (MapReduce) and storage (HDFS) components are running on the same machines; our focus is on disaggregated architecture. System model similar to ours is considered in [19], as well as in multiple industry approaches (e.g. [10]), however in these approaches index is stored in a key-value database, and as we show in this work it has two severe drawbacks comparing to our solution - significantly higher monetary cost and much longer load time.

To the best of our knowledge, our work is the first research attempt to improve queries performance in cloud data lakes by building indexing on top of object stores. However, two industry projects focus on the same domain - Microsoft Hyperspace [39] and Databricks Delta Lake [23]. Both assume data lakes architecture and try to improve queries performance by using indexes stored in object stores. Below we briefly discuss how these projects differ from our approach based on the available online information about these systems [23, 39].

- Hyperspace currently uses only "covering indexes" and plan to support other index types in the future. Covering indexes are built upon user-provided columns lists "indexed columns" and "data columns", and aim to improve the performance of the queries that search by "indexed columns" and retrieve "data columns". The implementation is based on copying both "indexed" and "data" columns from the data lake and storing them in a columnar format sorted by "indexed" columns. While covering indexes can improve some types of queries, they are not suitable for "needle in a haystack" queries where retrieval of all the columns should be supported (e.g. for GDPR scenario).
- Delta Lake uses Bloom filter indexes in the following way. Each file has an associated file that contains a Bloom filter containing values of indexed columns from this file. Then, upon a user query, Bloom filters are checked before reading the files, and the file is read only if the value was found in the corresponding Bloom filter. We use a similar technique (Section 3.4), but in our case, only a single Bloom filter will be read for a particular value, while in Delta Lake all existing Bloom filters should be read.

## 7 CONCLUSION

In recent years, cloud data lakes have become a prevalent way of storing large amounts of data. The main implication is a separation between the storage and compute layers, and as a result, a rise of new technologies in both compute and storage domains. Query engines, the main player in the compute domain, are designed to run on in-place data that is usually stored in a columnar format. Their main focus is on analytical queries while random reads are overlooked.

In this paper, we present our approach for optimizing *needle in a haystack* queries in cloud data lakes. The main idea is to maintain an index structure that maps indexed column values to their files. We built our solution in accordance with *data lake architecture*, where the storage is completely separated from the compute. The reason for building our index on top of object stores was motivated by the observation that the monetary cost of this solution is significantly lower than the one based on DBMS. Our experimental evaluation provides an additional reason to favor object stores over the DMBS approach - much lower index computation time.

For future work, we are planning to extend our index scheme and support more complex query types (e.g. joins, group by). An additional research direction may be an implementation of different systems according to data lake architecture and thereby improving their monetary cost and load time. For example, implementing a key-value store on top of object stores seems to be a promising research direction which can be based on the ideas presented in this work.

## REFERENCES

- [1] Daniel Abadi et al. 2015. SQL-on-hadoop systems: tutorial. *Proceedings of the VLDB Endowment* 8, 12 (2015), 2050–2051.
- [2] Daniel Abadi et al. 2020. The Seattle Report on Database Research. *ACM SIGMOD Record* 48, 4 (2020), 44–53.
- [3] Amazon. 2020. *Amazon Athena*. Retrieved November 14, 2020 from <https://aws.amazon.com/athena>
- [4] Amazon. 2020. *Amazon DynamoDB On-Demand Pricing*. Retrieved November 14, 2020 from <https://aws.amazon.com/dynamodb/pricing/on-demand/>
- [5] Amazon. 2020. *Amazon EC2 On-Demand Pricing*. Retrieved November 14, 2020 from <https://aws.amazon.com/ec2/pricing/on-demand/>
- [6] Amazon. 2020. *Amazon S3 pricing*. Retrieved November 14, 2020 from <https://aws.amazon.com/s3/pricing/>
- [7] Amazon. 2020. *AWS Glue*. Retrieved November 14, 2020 from <https://aws.amazon.com/glue>
- [8] Amazon. 2020. *Best Practices for Amazon EMR*. Retrieved November 14, 2020 from <https://d0.awsstatic.com/whitepapers/aws-amazon-emr-best-practices.pdf>
- [9] Amazon. 2020. *Best Practices for Running Apache Cassandra on Amazon EC2*. Retrieved November 14, 2020 from <https://aws.amazon.com/blogs/big-data/best-practices-for-running-apache-cassandra-on-amazon-ec2/>
- [10] Amazon. 2020. *Building and Maintaining an Amazon S3 Metadata Index without Servers*. Retrieved November 14, 2020 from <https://aws.amazon.com/blogs/big-data/building-and-maintaining-an-amazon-s3-metadata-index-without-servers/>
- [11] Amazon. 2020. *DynamoDB*. Retrieved November 14, 2020 from <https://aws.amazon.com/dynamodb/>
- [12] Amazon. 2020. *S3*. Retrieved November 14, 2020 from <https://aws.amazon.com/s3/>
- [13] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 International Conference on Management of Data (SIGMOD)*. 1383–1394.
- [14] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE micro* 23, 2 (2003), 22–28.
- [15] Andrzej Bialecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. 2012. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*. 17.
- [16] Spyros Blanas, Jignesh M Patel, Vuk Ercegovic, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD)*. 975–986.
- [17] B. H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
- [18] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, et al. 2019. Apache hive: From mapreduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 1773–1786.
- [19] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. 2013. Web data indexing in the cloud: efficiency and cost reductions. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*. 41–52.
- [20] B Barla Cambazoglu et al. 2013. A term-based inverted index partitioning model for efficient distributed query processing. *ACM Transactions on the Web (TWEB)* 7, 3 (2013), 1–23.
- [21] Samy Chambi et al. 2016. Better bitmap performance with roaring bitmaps. *Software: practice and experience* 46, 5 (2016), 709–719.
- [22] Fay Chang et al. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
- [23] Databricks. 2020. *Delta Lake*. Retrieved November 14, 2020 from <https://docs.databricks.com/delta>
- [24] Databricks. 2020. *Top 5 Reasons for Choosing S3 over HDFS*. Retrieved November 14, 2020 from <https://databricks.com/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html>
- [25] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
- [26] Jens Dittrich et al. 2010. Hadoop++ making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 515–529.
- [27] Xin Dong and Alon Halevy. 2007. Indexing dataspace. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD)*. 43–54.
- [28] Avriilia Floratou et al. 2014. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1295–1306.
- [29] Corinna Giebler et al. 2019. Leveraging the Data Lake: Current State and Challenges. In *International Conference on Big Data Analytics and Knowledge Discovery (DaWaK)*. Springer, 179–188.
- [30] Paolo Giudice et al. 2019. An approach to extracting complex knowledge patterns among concepts belonging to structured, semi-structured and unstructured sources in a data lake. *Information Sciences* 478 (2019), 606–626.
- [31] Google. 2020. *Google Cloud Storage*. Retrieved November 14, 2020 from <https://cloud.google.com/storage>
- [32] Michael Hausenblas and Jacques Nadeau. 2013. Apache drill: interactive ad-hoc analysis at scale. *Big Data* 1, 2 (2013), 100–104.
- [33] Marcel Kornacker et al. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proceedings of the 7th Conference on Innovative Data Systems Research (CIDR)*, Vol. 1. 9.
- [34] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
- [35] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.
- [36] Sergey Melnik et al. 2011. Dremel: interactive analysis of web-scale datasets. *Commun. ACM* 54, 6 (2011), 114–123.
- [37] Microsoft. 2020. *Azure Blob Storage*. Retrieved November 14, 2020 from <https://azure.microsoft.com/services/storage/blobs/>
- [38] Microsoft. 2020. *Azure subscription and service limits, quotas, and constraints*. Retrieved November 14, 2020 from <https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/azure-subscription-service-limits>
- [39] Microsoft. 2020. *Hyperspace*. Retrieved November 14, 2020 from <https://microsoft.github.io/hyperspace>
- [40] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database management systems*. McGraw-Hill.
- [41] Franck Ravat and Yan Zhao. 2019. Data lakes: Trends and perspectives. In *International Conference on Database and Expert Systems Applications (DEXA)*. Springer, 304–313.
- [42] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezh Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *Proceedings of the 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.
- [43] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–10.
- [44] Junjay Tan et al. 2019. Choosing a cloud DBMS: architectures and tradeoffs. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2170–2182.
- [45] IT Governance Privacy Team. 2020. *EU General Data Protection Regulation (GDPR)—An implementation and compliance guide*. IT Governance Ltd.
- [46] Grisha Weintraub. 2020. *data-lake-index*. Retrieved November 14, 2020 from <https://github.com/grishaw/data-lake-index>
- [47] Matei Zaharia et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.
- [48] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM computing surveys (CSUR)* 38, 2 (2006), 6–es.