Columnar Storage Optimization and Caching for Data Lakes

Guodong Jin*
jinguodong@ruc.edu.cn
Renmin University of China
Beijing, China

Haoqiong Bian*
haoqiong.bian@epfl.ch
EPFL
Lausanne, Switzerland

Yueguo Chen, Xiaoyong Du
{chenyueguo,duyong}@ruc.edu.cn
Renmin University of China
Beijing, China

ABSTRACT

As a unified data repository, data lake plays a vital role in enterprise data management and analysis. It composes the raw files into tables that are processed in-situ by various computation engines and applications. Therefore, the read performance of the tables is of great importance for analytical workloads in data lakes. In this paper, we improve the read performance from two dimensions: (1) storage-layout optimization that improves the I/O efficiency; (2) data caching that reduces the amount of I/Os. We observe that storage-layout optimization in existing work is limited by the physical row group boundary determined by data ingestion, while the existing caches in the software stack of data lakes are not dedicated to analytical queries on column stores. Therefore, we apply the inter-row-group layout optimization to overcome the former limitation and propose a columnar caching mechanism with a lazy replacing policy for analytical workloads. We also show initial evaluation results to support our design.

1 INTRODUCTION

A data lake is a unified data repository for analytical applications in a company or an organization. It provides table storage and management services on a vast amount of raw files stored in distributed file systems (e.g., HDFS) or cloud storage (e.g., AWS S3 [1]). The tables backed by raw files are processed in-situ by various query engines, avoiding the long-running and expensive ETL. Within a data lake, columnar file formats, such as Parquet [9] and ORC [7], are usually applied to improve the I/O efficiency of analytical queries.

In data lakes, structured data is ingested and encoded into row groups following the PAX table layout [19] , where columnar format is applied in the row group. As shown in Figure 1, the data within each row group is organized in columns that are stored sequentially. We refer to each column of a row group as a *column chunk*, which is independently compressed with a domain-specific compression algorithm (e.g., dictionary encoding, run-length encoding) to reduce space overheads and I/O costs. Given a query, as projection is pushed down to the table scan operator, it only reads needed column chunks.

Although it is confirmed that the PAX table layout is efficient for data lakes, and packing columns into separate files (i.e., pure columnar table layout) is unnecessary [15, 19], storage layout in data lakes is still far from optimal. Many existing studies improve the read performance of the PAX table layout from two major dimensions: *storage-layout optimization* that improves the I/O efficiency; *data caching* that reduces the amount of I/Os.

Storage-layout Optimization. To reduce the disk seek cost of reading the needed columns, column ordering [14] reorders the physical positions of column chunks within the same row group and puts frequently co-accessed column chunks into nearby positions. Such a way of physical storage-layout optimization is especially effective for wide tables with many columns, as random disk seeks take a large proportion of the I/O cost when reading a few from the many column chunks in a row group of fixed size (typically tens or a few hundreds of megabytes [6, 8]). Wide tables are prevalent in today's data lakes [4, 12, 14], and are used to convert the complex and expensive distributed joins into simple table scans. In addition to column ordering, increasing row group size can further improve the I/O performance [13, 14]. However, increasing row group size will sacrifice the efficiency and timeliness of data ingestion [13, 14].

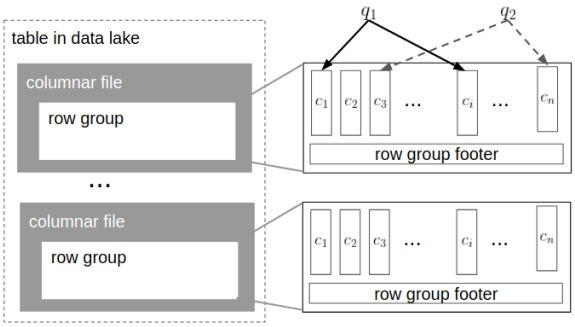

Figure 1: An illustration of the table layout in columnar file format, where c_i is a column chunk in the row group and q_i is a query.

Data Caching. In addition to storage-layout optimization, caching is applied in various layers of a data lake, such as page cache in the Linux kernel, file cache in the storage system (e.g., HDFS Centralized Cache [2]), distributed file caching system (e.g., Alluxio [17]), and intermediate result cache in the query engine (e.g., Spark RDD Cache [11]). However, each existing cache has its limitation. The page cache employs LRU-based policies, and the query engine cannot directly control which part of the data is cached. The hot data that is more worthy of being cached might be evicted by a query that reads a large amount of cold data. The file cache in storage systems and the distributed file caching systems cache entire files or blocks. For PAX layout that stores all the column chunks of a row group in the same file or block, they lead to very low space efficiency. For the intermediate result caching in a specific query engine, it cannot be shared by queries and applications running outside of the engine. Moreover, the cache is lost when the query session is closed or the query engine is shut down.

In this paper, we present a column storage engine with (1) storage-layout optimization across row group boundary without affecting data ingestion; (2) a lazy columnar cache. For storage-layout optimization, we apply asynchronous and inter-row-group column compaction on the recently ingested row groups to improve query performance without affecting the efficiency of data ingestion. For data caching, we choose column chunk as the best cache granularity for analytical queries and enable efficient

* Haoqiong Bian and Guodong Jin contributed equally to this work.

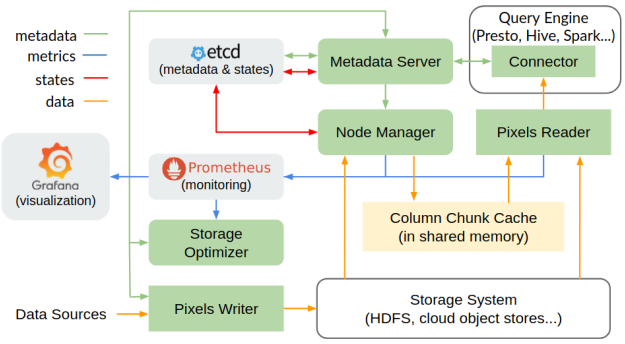

Figure 2: The architecture of Pixels.

cache accesses. We also envisage that lazy cache replacing (i.e., replace the cache according to long-term workload patterns) provides a higher hit rate than the eager replacing policies (i.e., replace the cache whenever a data item is accessed).

We have implemented the storage-layout optimization and the columnar caching framework in our column storage engine, Pixels[1]. In the rest of this paper, we introduce the architecture of Pixels in Section 2, present storage-layout optimization and columnar caching in Section 3 and 4, respectively, and show evaluation results in Section 5.

2 PIXELS ARCHITECTURE

The architecture of Pixels is shown in Figure 2. It consists of six main components (the light green blocks).

Metadata Server stores and maintains the system metadata, including table schema and data location, in Etcd [5], which is a distributed and high-available key-value store.

Storage Optimizer is a background process that monitors performance metrics and the evolution of query workload. It adaptively generates new storage layouts and cache plan (i.e., column chunks to be cached on each node). The new data layout and cache plan are maintained by *Metadata Server* and will be applied to the upcoming ingested data.

Node Manager reports each node's hardware metrics, such as CPU, memory, disk, and network usages to Prometheus [10] for performance profiling and storage optimization. It is also responsible for replacing the *Column Chunk Cache* according to the update of the cache plan.

Pixels Reader and Writer are the I/O interfaces in the client libraries used to read and write files in Pixels format. The reader probes the *Column Chunk Cache* transparently.

Connector is the library used by query engines to connect to Pixels. It requests metadata from *Metadata Server* for query parsing and is responsible for generating query splits for table scan and calling *Pixels Reader* to read data. The connectors for various query engines are similar, thus developing a new connector requires few efforts.

Scalability and Fault-tolerance. In the architecture, *Metadata Server* and *Node Manager* run independently as stateless daemons, without explicit inter-node state synchronization. All states are maintained in Etcd [5] as it is fault-tolerant and reliable. If any daemon crashes, it can be immediately restarted by the guard process, without an expensive recovery progress. This makes Pixels scalable and easy to deploy and maintain.

[1]https://github.com/pixelsdb/pixels

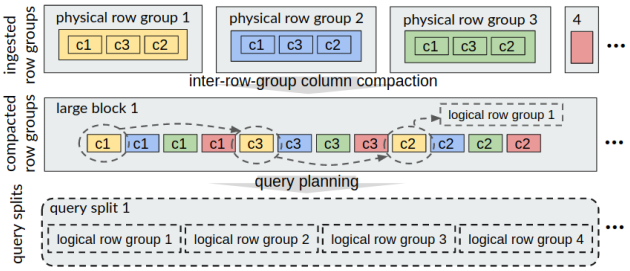

Figure 3: Inter-row-group column compaction example.

3 STORAGE-LAYOUT OPTIMIZATION

In this paper, we take HDFS as the underlying storage. From the vertical dimension, by putting the co-accessed column chunks within a row group to adjacent or closer positions (i.e., intra-row-group column ordering), the disk seek distance to read the column chunks is eliminated or reduced. As seek cost (which can dominate the I/O cost of a query) and seek distance are positively correlated [14], the I/O efficiency can be improved by intra-row-group column ordering.

From the horizontal dimension, increasing row group size, i.e., reducing the number of row groups in a table of a given size, also improves I/O efficiency, because it reduces the number of seek-and-read operations required to scan the table. However, row group size is limited by data ingestion [13], during which, data is consumed, encoded, and packed into in-memory row groups and then written as blocks into the storage. As data ingestion is CPU intensive, it is executed using many threads (cores) to ingest the fast arriving data in time. Each thread produces a row group at a time using limited memory space. Typically, the size of a row group in the storage (after encoding and serialization) is only tens of megabytes [6, 8]. For the prevalent wide tables (e.g., with 1000 columns) in data lakes, the average size of the column chunks is relatively small (e.g., tens of kilobytes). Therefore, even with column ordering, the I/O pattern of wide-table scan is still dominated by random access, resulting in sub-optimal query performance. [13] improves I/O performance by increasing the row group size but sacrifices the data ingestion efficiency.

3.1 Inter-row-group Column Compaction

To further improve I/O performance without affecting the efficiency of data ingestion, we propose *inter-row-group column compaction*. As illustrated by the example in Figure 3, four physical row groups produced in data-ingestion stage (where intra-row-group column ordering is applied to tune the column order) are *asynchronously* compacted into a large HDFS block. Within the block, column chunks of the same column are put together while the order between columns remains the same as in the physical row groups. Metadata footers of the physical row groups are packed into the footer of the large block. Column-chunk pointers in the row group footers are modified to point them to the new positions after compaction. Row groups still exist logically. During query planning, logical row groups within the same block are packed into the same query split that is scanned by a single table scan task. In this way, as the query can scan the column chunks of the same column sequentially without multiple disk-seeks in HDFS, the I/O performance is significantly improved. The efficiency of data ingestion is not affected either, as column compaction is performed asynchronously. The recently ingested (physical) row groups that have not been compacted

are also involved in query execution to reflect the freshest data. In practice, we compact 2^i (e.g., 32) row groups into one block. It leads the block to be 1-2GB, which is appropriate for HDFS.

Although adaptively packing data units into pages or blocks according to workload type (e.g., OLTP and OLAP) has been studied in related works [18, 20], to the best of our knowledge, compacting column chunks acrossing row group boundaries has not been proposed to improve the I/O efficiency in column stores.

If the underlying storage is a cloud object store (e.g., AWS S3 [1]), the disk seek latency is then replaced by remote access latency. Although their cost models are different, we observe that putting co-accessed column chunks closer and fetching them by a single request can also help improve the performance. Storage-layout optimization for cloud storage is our in-progress work.

4 COLUMNAR CACHING

In this section, we describe the columnar caching mechanism in Pixels, which takes column chunk as an atomic unit of data access and eviction, and applies a lazy replacing policy instead of the LRU or LFU variants that eagerly replace the cache content upon cache misses. As discussed in Section 2, the Storage Optimizer generates the global cache plan that is stored in etcd and specifies the files (sub-plan) to be cached on each node. The Node Manager on each node is responsible for monitoring and applying the changes in the sub-plan of the node. The set of column chunks to be cached is the same for all the row groups, thus the cache plan is light-weight and realistic for large setups.

4.1 Read-Optimized Cache Access Protocol

We propose a read-optimized access protocol for the analytical queries to access the cache on each node efficiently. As illustrated in Figure 4, on each node, the column chunks are cached and indexed by an adaptive radix tree [16]. Cache misses are recorded in a local message queue (MQ). These structures are in the shared memory and can be directly accessed by the query.

To control the concurrent reads and writes on the cache, we use *rw_flag* to indicate if the cache is being written or not, and *reader_cnt* to record the read concurrency. They are the only synchronization points between the read and the write (i.e., replacement) operations, and are packed into a 32-bit integer that can be operated atomically by a single memory instruction (i.e., lock-free). Each replacement first sets *rw_flag* and waits util *reader_cnt* becomes zero. For each cache read, the reader first (R1) increases *reader_cnt* if *rw_flag* is false and then looks up the column chunk in the index. Otherwise, it waits for the concurrent replacement. If the column chunk is hit in the index, the reader (R2) reads the column chunk from the cache area and increases *hit_cnt*. Otherwise, it (R3) sends a cache miss message to the MQ. When the read is finished, *reader_cnt* is decremented.

As the *Node Manager* is protected by its guard (Section 2), the replacement always completes and resets *rw_flag*. Meanwhile, a timeout mechanism is applied to abort the abandoned reads and resets *reader_cnt*, avoiding endlessly blocking the replacement.

Cache replacement happens when there is a new cache plan in etcd (Figure 2). It has to read the new column chunks from underlying file systems, which is time-costing. To reduce the blocking of reads, we divide the replacement into three steps: (W1) Pack the survived column chunks into the front of the cache area and update the index. It is very fast as no I/Os are involved; (W2) Read the new column chunks into the remaining cache area in the background without blocking cache reads; (W3) Update the index to make the new column chunks visible to queries. As the cache is read in a lock-free fashion and the read-blocking phase in the replacement is drastically reduced, this access protocol is efficient for reads.

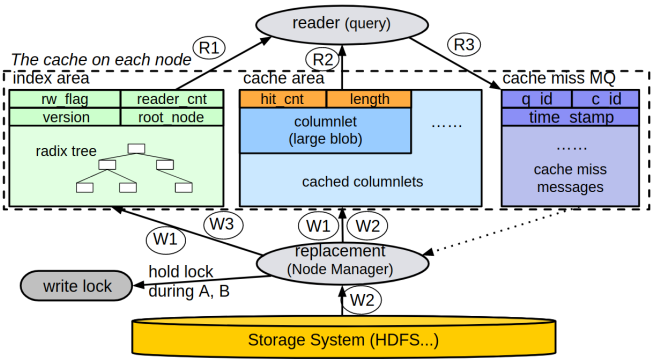

Figure 4: Illustration of the data structure and access protocol of the column chunk cache on each node.

4.2 Lazy Cache Replacement

In Pixels, we envisage the high efficiency of **lazy** cache replacement. The cache misses in each node are collected by *Prometheus* (Section 2), without immediately triggering cache replacement. Whenever the workload pattern evolves, the *Storage Optimizer* calculates the *cache efficiency* of each column chunk by dividing its hit+miss count by its size. The hit count is stored before each column chunk in the cache area (Figure 4), while the miss count is collected from the cache miss messages in the cache miss MQ. The most-efficient column chunks that do not exceed the cache capacity are included in the new cache plan that is then applied across the cluster. This is based on the temporal locality of the column access.

Other specific algorithms, such as machine learning, can be used to predict the workload changes, although it is beyond the scope of this paper. When given a relatively limited memory space compared to the large on-disk column store, lazy cache replacement avoids repeated and inefficient cache swapping in and out. It is intuitive that caching the most frequently accessed column chunks according to the long-term workload provides a higher overall cache-hit rate.

5 EVALUATION

In the evaluation, we use Presto-0.192 as the query engine and HDFS-2.7.3 as the storage. The Java version is 1.8.0_172. The operating system is CentOS-7.8. The hardware platform is a 4-node cluster. Each node has 2×Xeon E5-2650-v3 CPU (20 cores / 40 threads in total, 2.30GHz), 256GB DRAM, 8×4TB HDD, 2×1Gbps NIC. HDFS Datanode, Presto Worker, and Pixels Node Manager are deployed on each node. Each HDFS Datanode uses all the 8 HDDs on the node. Each Presto Worker is allocated 172GB of memory and 32 threads.

Workload: We use a real-world workload from [14] for evaluations. The table contains 1187 columns. There are 547 unique queries in the workload, with 77.5% of the queries accessing 7 - 20 columns, and a small fraction of them accessing more than 100 columns. We generate 3.75TB of data (in CSV format) for the table, and the encoded data in Pixels format is 1.18TB. All the queries are in the following form:

```
SELECT ... FROM t [WHERE ...]
[GROUP BY ...] [ORDER BY ...]
```

Configurations: We evaluate the latency of all the 547 queries using Pixels under five different configurations: (1) the simulated Parquet storage layout without any caching (`Parquet-S`), where the column order is the same as the one in the table's schema. This is the storage layout used by Parquet [9] and ORC [7]. We have evaluated the performance of Parquet and Pixels on the aforementioned workload. Result shows that using this layout, Pixels is slightly faster than Parquet for most queries. We use this simulation in our experiments to exclude the performance variance caused by implementations; (2) intra-row-group column ordering without any caching (`IntraRG`); (3) inter-row-group column compaction without any caching (`InterRG`); (4) inter-row-group column compaction with Linux page cache (`PageCache`); (5) inter-row-group column compaction with our column chunk cache (`ColCache`). The column chunk cache or the page cache in each node is limited to 64GB. And the cache plan for column chunk cache is calculated based on the entire workload.

5.1 Inter-row-group Column Compaction

We use the query performance on Parquet-S as the baseline and compare Pixels with InterRG against IntraRG. The boxplots of Parquet-S, IntraRG, and InterRG are shown in Figure 5, while the latency percentiles of the queries are listed in Table 1.

From Figure 5, we can see that IntraRG significantly outperforms Parquet-S, while InterRG further reduces query latency by large margins over IntraRG. For the 25th percentile, median, and 75th percentile query latency, InterRG outperforms IntraRG by 5.0x (6.9s vs. 34.5s), 3.1x (13.9s vs. 43.4s) and 3.0x (18.0s vs. 53.5s), respectively. It shows that our layout optimization across row group boundaries can consistently provide better read performance on the basis of inter-row-group column ordering. The reason is that a query task can scan the column chunks from different row groups with sequential disk read. However, for queries that scan large amounts of data, the benefits of InterRG are limited because disk seek no longer dominates the I/O cost. For example, *q17* scans around 750GB data from 15 columns, and sequential scans dominate its cost. Thus InterRG improves very little compared to IntraRG (300.3s vs. 308.7s).

5.2 Columnar Caching

We compare column chunk cache with Linux page cache based on the InterRG layout. The boxplots of PageCache and ColCache are shown in Figure 5. As expected, both caching solutions yield much better performance than InterRG. However, ColCache further outperforms PageCache on most queries, especially for the queries at the end of the percentiles. We observe that ColCache outperforms PageCache by 1.4x (5.1s vs. 6.9s), 1.65x (6.5s vs. 10.7s), 2x (7.7s vs. 15.5s), and 1.8x (16.8s vs. 30.3s), respectively, in terms of the 75th, 85th, 90th, and 95th percentile query latency. It also reduces the maximum query latency from 325.6s to 289.6s.

For individual queries, ColCache is faster than PageCache on 74% of queries, by 1.61x on average and 7.24x maximum. PageCache is only obviously faster than ColCache on 16% of queries by 1.06x - 2.65x (1.22x on everage). The evaluation result confirms that the lazy cache replacement provides better overall query performance for analytical workloads than the LRU variant (which is eager) used in page cache. However, the ColCache configuration is an extremely lazy case, i.e., a single cache plan is used for the entire workload. Intuitively, adaptive replacement driven by the workload evolution should provide better query performance.

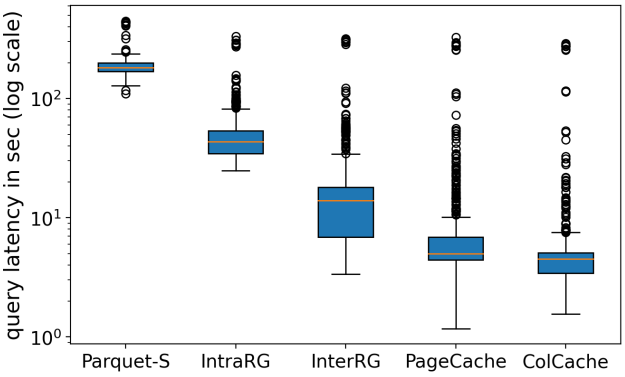

Figure 5: Query latency (in seconds) on Pixels under different configurations.

Table 1: Detailed percentiles of query latency (in seconds) on Pixels under different configurations.

	Min	5th	25th	50th	75th	95th	Max
Parquet-S	109.7	149.4	169.6	182.5	199.2	231.3	447.6
IntraRG	24.8	26.5	34.5	43.4	53.5	97.2	333.9
InterRG	3.4	5.1	6.9	13.9	18.0	57.9	317.2
PageCache	1.2	3.2	4.4	5.0	6.9	30.3	325.6
ColCache	1.6	3.1	3.4	4.5	5.1	16.8	289.6

6 CONCLUSION

In this paper, we propose the storage-layout optimization and columnar caching solution that improve the I/O efficiency for columnar storage in data lakes. Motivated by the observation that existing layout optimizations are limited by the physical row group boundary determined by data ingestion, we presented a novel inter-row-group column compaction to overcome the limitation and further improves I/O efficiency. Besides, we observed that existing caches in the software stack of data lakes are not dedicated to analytical queries on column stores, thus, we also proposed a columnar caching mechanism with a lazy replacing policy. We presented initial experimental results demonstrating that our inter-row-group column compaction consistently yields better performance than existing layout optimizations, and our columnar caching improves the performance of almost all queries on top of inter-row-group column compaction, and outperforms Linux page cache on most queries. All of our works are implemented inside a column storage engine named *Pixels*.

In the future, we plan to continue working as following: (1) Pixels has a plug-able interface for underlying storage systems. In addition to HDFS, we are working on storage optimization for other distributed file systems (e.g., Ceph [3]) and cloud stores (e.g., AWS S3 [1]). (2) The efficiency of columnar caching with extremely lazy replacement policy has been confirmed. In the next step, we are going to explore the solutions for the lazy replacement that adapts to the trend of workload evolution.

ACKNOWLEDGMENTS

This work has been partially funded by the National Key Research and Development Program of China (2020YFB1710004), the European Unions Horizon 2020 research and innovation programme under the grant agreement No 825041 (SmartDataLake), and NSFC grant No. U1711261. Yueguo Chen is the corresponding author.

REFERENCES

[1] 2022. *Amazon S3.* https://aws.amazon.com/s3/

[2] 2022. *Centralized Cache Management in HDFS.* https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html

[3] 2022. *Ceph Documentation.* https://docs.ceph.com/en/pacific/

[4] 2022. *Clickhouse Features.* https://clickhouse.com/docs/en/

[5] 2022. *Etcd Home Page.* https://etcd.io/

[6] 2022. *Orc Configuration. https://orc.apache.org/docs/hive-config.html.* https://orc.apache.org/docs/hive-config.html

[7] 2022. *ORC. https://orc.apache.org/.* https://orc.apache.org/

[8] 2022. *Parquet Configuration. http://parquet.apache.org/documentation/latest/.* http://parquet.apache.org/documentation/latest/

[9] 2022. *Parquet. http://parquet.apache.org/.* http://parquet.apache.org/

[10] 2022. *Prometheus.* https://prometheus.io/

[11] 2022. *RDD Programming Guide.* https://spark.apache.org/docs/latest/rdd-programming-guide.html

[12] 2022. *Uber's Big Data Platform: 100+ Petabytes with Minute Latency.* https://eng.uber.com/uber-big-data-platform/

[13] Haoqiong Bian, Youxian Tao, Guodong Jin, Yueguo Chen, Xiongpai Qin, and Xiaoyong Du. 2018. Rainbow: Adaptive Layout Optimization for Wide Tables. In *ICDE*.

[14] Haoqiong Bian, Ying Yan, Wenbo Tao, Liang Jeff Chen, Yueguo Chen, Xiaoyong Du, and Thomas Moscibroda. 2017. Wide Table Layout Optimization Based on Column Ordering and Duplication. In *SIGMOD*.

[15] Yin Huai, Siyuan Ma, Rubao Lee, Owen O'Malley, and Xiaodong Zhang. 2014. Understanding Insights into the Basic Structure and Essential Issues of Table Placement Methods in Clusters. *PVLDB* 6, 14.

[16] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*.

[17] Haoyuan Li. 2018. *Alluxio: A Virtual Distributed File System.* Ph.D. Dissertation. UC Berkeley.

[18] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *SIGMOD*.

[19] Armbrust Michael, Das Tathagata, Sun Liwen, Yavuz Burak, Zhu Shixiong, Murthy Mukul, Torres Joseph, Hovell Herman, van, Ionescu Adrian, uszczak Alicja, Switakowski Micha, Micham Szafranski, Li Xiao, Ueshin Takuya, Mokhtar Mostafa, Boncz Peter, Ghodsi Ali, Paranjpye Sameer, Senster Pieter, Xin Reynold, and Zaharia Matei. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. In *VLDB*.

[20] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-store: A real-time OLTP and OLAP system. In *EDBT*.