AnalyticDB: Real-time OLAP Database System at Alibaba Cloud

Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin,
Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, Chengliang Chai

Alibaba Group

{lizhe.zcq,maomeng.smm,chuangxian.wcx,xiaoqiang.pxq,yibo.ll,sh.wang,chenzhe.cz,
lifeifei,itlanger.pany,f.zheng,chengliang.ccl}@alibaba-inc.com

ABSTRACT

With data explosion in scale and variety, OLAP databases play an increasingly important role in serving real-time analysis with low latency (e.g., hundreds of milliseconds), especially when incoming queries are complex and ad hoc in nature. Moreover, these systems are expected to provide high query concurrency and write throughput, and support queries over structured and complex data types (e.g., JSON, vector and texts).

In this paper, we introduce AnalyticDB, a real-time O-LAP database system developed at Alibaba. AnalyticDB maintains all-column indexes in an asynchronous manner with acceptable overhead, which provides low latency for complex ad-hoc queries. Its storage engine extends hybrid row-column layout for fast retrieval of both structured data and data of complex types. To handle large-scale data with high query concurrency and write throughput, AnalyticDB decouples read and write access paths. To further reduce query latency, novel storage-aware SQL optimizer and execution engine are developed to fully utilize the advantages of the underlying storage and indexes. AnalyticDB has been successfully deployed on Alibaba Cloud to serve numerous customers (both large and small). It is capable of holding 100 trillion rows of records, i.e., 10PB+ in size. At the same time, it is able to serve 10m+ writes and 100k+ queries per second, while completing complex queries within hundreds of milliseconds.

1. INTRODUCTION

AnalyticDB is an OLAP database system designed for high-concurrency, low-latency, and real-time analytical queries on PB scale, and has been running on 2000+ physical machines on Alibaba Cloud [1]. It serves external clients on

Alibaba Cloud from a wide range of business sectors, including e-commerce, finance, logistics, public transit, meteorological analysis, entertainment, etc., as well as internal business operations within Alibaba Group.

Recent works [35, 28, 29, 36, 25] have summarized the main challenges of designing an OLAP system as achieving low query latency, data freshness, flexibility, low cost, high scalability, and availability. Compared to these works, analytical workloads from our clients elevate AnalyticDB to an even larger scale: 10PB+ data, hundred thousands of tables and trillions of rows, which presents significant challenges to the design and implementation of AnalyticDB.

The first challenge is that today's users face more complicated analytics scenarios than before, but still have high expectation for low query latency. Users often do not tolerate queries that spend a long time. However, as users of AnalyticDB come from various domains, their analytical demands differ significantly and may change frequently, which make their diverse and complex queries hard to optimize. The queries range from full scan, point lookup to multitable join, and involve conditions on many combinations of columns. Although indexing is a straightforward way to improve query performance, building indexes on pre-specified columns is often no longer effective.

The second challenge is that emerging complex analysis tends to involve different types of queries and data at the same time, which requires the system to have a friendly and unified data layout at the storage layer. Traditional O-LAP queries and point-lookup queries require different layouts, i.e., column-stores and row-stores respectively [34, 12]. Furthermore, more than half of our users' data has a complex data type, such as text, JSON string, vector, and other multi-media resources. A practical storage layout should be able to provide fast retrieval for many data types, in order to efficiently support queries involving both structured data and data of complex types.

The third challenge is that while the system is processing real-time queries with low latency, it also needs to handle tens of millions of online write requests per second. Traditional designs [6, 8, 10, 29, 5] process reads and writes in the same process path, so that reads could see newly written data once it is committed. However, such designs are no longer well-suited for our case, as consuming a large portion of resources to guarantee query performance will hurt write performance, and vice versa. Careful designs to balance among query latency, write throughput and data visibility should be taken into consideration.

To address above challenges, we propose and implement several novel designs in AnalyticDB and have made the following contributions:

Efficient Index Management. AnalyticDB embeds an efficient and effective index engine, which leverages two key approaches for achieving low latency with acceptable overhead. First, indexes are built on all columns in each table for significant performance gain on ad-hoc complex queries. We further propose a runtime filter-ratio-based index path selection mechanism to avoid performance slow-down from index abuse. Second, since it is prohibitive to update large indexes in the critical path, indexes are asynchronously built during off-peak periods. We also maintain a lightweight sorted-index to minimize the impact of asynchronous index building on queries involving incremental (i.e., newly-written after index building has started) data.

Storage layout for structured data and data of complex types. We design the underlying storage to support hybrid row-column layout. In particular, we utilize fast sequential disk IOs so that its overhead is acceptable under either OLAP-style or point-lookup workloads. We further incorporate complex data types in the storage (including indexes) to provide the capability of searching resources (i.e., JSON, vector, text) together with structured data.

Decoupling Read/write. In order to support both high-throughput writes and low-latency queries, our system follows an architecture that decouples reads and writes, i.e., they are served by write nodes and read nodes respectively. These two types of nodes are isolated from each other and hence can scale independently. In particular, write nodes persist write requests to a reliable distributed storage called Pangu [7]. To ensure data freshness when serving queries, a version verification mechanism is applied on read nodes, so that previous writes processed on write nodes are visible.

Enhanced optimizer and execution engine. To further improve query latency and concurrency, we enhance the optimizer and execution engine in AnalyticDB to fully utilize the advantages of our storage and indexes. Specifically, we propose a storage-aware SQL optimization mechanism that generates optimal execution plans according to the storage characteristics, and an efficient real-time sampling technique for cardinality estimation in cost based optimizer. Besides, we design a high-performance vectorized execution engine for the hybrid storage that improves the efficiency of computation-intensive analytical queries.

The rest of the paper is organized as follows. Section 2 discusses related work and Section 3 presents read/write decoupling architecture. The data structures and indexes for structured data and complex data types are described in Section 4, and optimizations on optimizer and execution engine are described in Section 5. Section 6 evaluates AnalyticDB and Section 7 concludes the paper.

2. RELATED WORK

AnalyticDB is built from scratch for large-scale and real-time analysis on cloud platform. In this section, we compare AnalyticDB with other systems.

OLTP databases. OLTP databases such as MySQL [6] and PostgreSQL [8] are designed to support transactional queries, which can be considered as point lookups that involve one or several rows. Hence, storage engines in OLTP databases are row-oriented and build B+tree index [16] to speed up query performance. However, row-store does not

fit for analytical queries as these queries only require a subset of columns, where row-store significantly amplifies I/Os. Moreover, OLTP databases usually actively update indexes in the write path, which is so expensive that affects both write throughput and query latency.

OLAP databases. To improve the efficiency of analytical queries, many OLAP databases like Vertica [29], Teradata DB [10] and Greenplum [5] have been developed. Vertica [29] uses projection to improve query performance. Instead of building conventional indexes on columns, it only keeps information about Min/Max values, leading to high latency from less effective pruning. Teradata DB [10] and Greenplum [5] adopt column-store and allow users to specify indexed columns. However, they have two main limitations: first, they modify column indexes in the write path, which is prohibitive for all-column indexes; second, column-store requires many random I/Os for point-lookup queries.

Big Data systems. With the emergence of Map-Reduce model [18], batch processing engines, such as Hive [35] and Spark-SQL [37, 13], become popular to process large-scale data on many machines. However, the executed queries are considered "offline". The whole execution could last for minutes or hours, and thus is not well-suited for real-time queries. Impala [28] converts "offline" queries to interactive queries using pipeline processing model and column-store, reducing latency of common queries to seconds. However, Impala does not have column indexes (with Min/Max statistics only), so that it can not handle complex queries well.

Real-time OLAP systems. Recent real-time OLAP systems include Druid [36] and Pinot [25], both adopting column store. They all build bitmap-based inverted indexes, i.e., Pinot on all columns and Druid on Dimension Columns. A query on Druid may suffer high latency, if it involves columns not in Dimension Columns. All their indexes are updated in the write path, which affects the write performance. Besides, they lack support for some important query types such as `JOIN`, `UPDATE`, and `DELETE`, and are inefficient for point-lookup queries as they are column-oriented.

Cloud Analytical Services. A number of cloud analytical services such as Amazon Redshift [21] and Google BigQuery [33] have recently emerged. Amazon Redshift is a fully managed cloud database service. It uses columnar storage and massively parallel processing (MPP) to distribute queries across multiple nodes. A typical Redshift cluster has two or more Compute Nodes which are coordinated through a Leader Node. Compared to it, AnalyticDB introduces a read/write decoupling architecture with independent read nodes and write nodes, as well as a set of coordinators to communicate with. Google BigQuery is the external implementation of one of Google's core technologies called Dremel [31], which includes columnar storage for high storage efficiency and a tree topology for dispatching queries and aggregating results across thousands of machines in seconds. This is different from AnalyticDB, which leverages an efficient index engine and a DAG execution framework.

3. SYSTEM DESIGN

As a cloud database, AnalyticDB runs on top of Apsara, a large-scale, general-purpose and highly-reliable computing infrastructure developed at Alibaba Cloud since 2009. Apsara manages all resources among hundreds of thousands of physical machines, and sustains many Alibaba Cloud services, such as searching, computing and storage. AnalyticD-
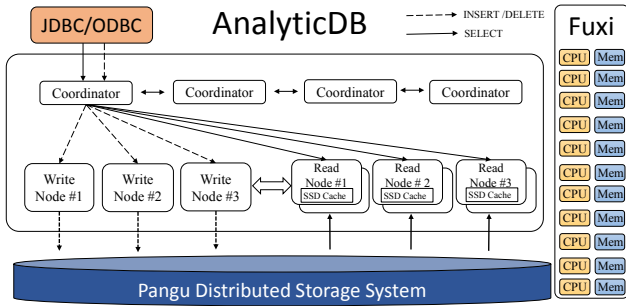
Figure 1: Architecture of AnalyticDB

```
CREATE TABLE db_name.table_name (
    id int,
    city varchar,
    dob date,
    primary key (id)
)
PARTITION BY HASH KEY(id)
PARTITION NUM 50
SUBPARTITION BY LIST (dob)
SUBPARTITION OPTIONS (available_partition_num = 12);
```

Figure 2: DDL for creation of partitioned table.

B leverages two core components in Apsara, i.e., Pangu [7] (a reliable distributed storage system) and Fuxi [38] (a resource manager and job scheduler), as shown in Figure 1. In this section, we present the vital design choices made in AnalyticDB, including data model and system architecture.

3.1 Data Model and Query Language

AnalyticDB follows a standard relational data model, i.e., records in tables with fixed schema. In addition, many popular complex data types, e.g., full-text, JSON and vector, are also supported to fulfill increasing analytical demands from real-world applications (detailed in Section 4.1.1 and 4.2.2). AnalyticDB supports ANSI SQL:2003 [19] and enhances it with additional features, such as partition specifications and complex-typed data manipulations.

3.2 Table Partitioning

In AnalyticDB, each table is subjective to two levels of partition, i.e., primary and secondary. Figure 2 illustrates a sample DDL that creates a table with two-level partition, i.e., a primary partition on column *id* with 50 partitions and a secondary partition on *dob* with 12 partitions. The primary partition is based on the hash of a user-specified column, and hence rows are distributed among all primary partitions to maximize concurrency. In practice, any column with high cardinality could be chosen as this partition column, which makes each partition balanced. Besides, users can optionally specify a secondary partition (called *subpartition*). The secondary partition is a list partition with a threshold defining the maximum partition number, for automatic data retention and recycling. Usually, a column representing time intervals (e.g., day, week or month) is chosen as the secondary partition column, and hence rows within the same interval are grouped as a partition. Once the number of partitions exceeds the threshold, the oldest partition is discarded automatically.

3.3 Architecture Overview

Figure 1 shows the system architecture. In overall, there are mainly three types of nodes in AnalyticDB, i.e., Coor-
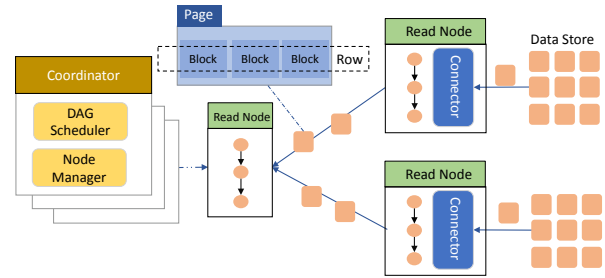

Figure 3: Pipeline-mode execution engine.

dinator, Write Node and Read Node. Coordinators collect requests (both writes and queries) from clients through JDBC/ODBC connections, and dispatches them to corresponding write nodes and read nodes. Write nodes are responsible for processing writes (such as INSERT, DELETE, UPDATE), and flush SQL statements into Pangu for persistence. Read nodes are responsible for handling queries (such as SELECT). In this manner, Write nodes and read nodes are decoupled from each other (detailed in Section 3.4). Fuxi utilizes available resources in all these nodes to provide computing workers for asynchronous task execution. In addition, AnalyticDB provides a general-purpose and pipeline-mode execution engine (shown in Figure 3) that runs on computing workers. Data flows through the system in units of column blocks (called *Page*s) from the storage to the client. All data processes are in memory and are pipelined between different stages across the network. This pipeline workflow enables AnalyticDB to serve users' complex queries with high throughput and low latency.

3.4 Read/Write Decoupling

Traditional OLAP databases couple reads and writes, i.e., a database instance executes any arrived request in the same execution path, without considering whether it is a read or write. Therefore, all concurrent requests share a resource pool and hence affect each other. In the scenario where both query and write concurrency is high, this design leads to poor performance from resource contention. To address this issue, we propose a read/write decoupling architecture. Write nodes are only responsible for write requests while read nodes are for queries. These two types of nodes are isolated from each other, so that writes and queries are handled in completely different execution paths.

3.4.1 High-throughput Write

One of write nodes is selected as master and others as workers, and they coordinate with each other through lock service built on ZooKeeper [24]. When write nodes are first launched, the master configures table partitioning (Section 3.2) on workers. Coordinators can then distribute write requests to corresponding workers based on this configuration. When a write request arrives, the coordinator first parses SQL and recognizes it as a write, and then dispatches it to the corresponding write node. Each write node works as an in-memory buffer for received SQL statements and periodically flushes them as a log to Pangu (similar to log writer threads in traditional databases). Once the buffer is completely flushed, the node returns a version (i.e., log sequence number) to coordinators, which then return users a success message for each committed write.

When the number of log files on Pangu reaches a certain scale, AnalyticDB will launch multiple MapReduce [18] jobs
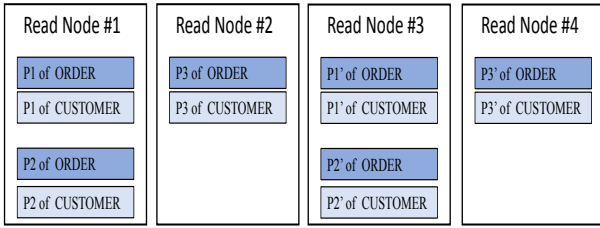
Figure 4: Data placement among read nodes.

on Fuxi to convert log commits into actual data files, i.e., baseline data and indexes (detailed in Section 4).

3.4.2 Real-time Read

Each read node is assigned a number of partitions by co-ordinators, where partitions with the same hash value are placed in one node. Figure 4 shows this partition placement among read nodes, With the storage-aware optimizer (Section 5.1), this placement helps to save the cost of data redistribution by more than 80%, measured from our production service. Moreover, read nodes are replicated for concurrency and reliability (detailed in Section 3.4.3). Each node loads initial partitions from Pangu, and pulls subsequent updates from corresponding write nodes periodically. It applies updates to its local data copies, which are not written back to Pangu. We choose to continuously pull data from write nodes instead of Pangu, in order to reduce synchronization latency. Therefore, The write node performs as a cache to serve concurrent pulls from different read node replicas.

Since recent writes need be remotely fetched, read nodes provide two visibility levels to users: *real-time* read where data can be immediately read after written; and *bounded-staleness* read where data is visible within a bounded delay. To sustain low query latency, AnalyticDB uses bounded-staleness read by default, which is acceptable in most O-LAP scenarios. For users demanding high visibility, real-time read can be enabled, which however exposes a data synchronization issue between read nodes and write nodes.

To solve this issue, we use a version verification mechanism. Specifically, each primary partition is associated with its own version on write node. After a number of write requests on a partition are flushed, the write node will increase this partition's version and attach this value to the response message. Let's look at Figure 5 and take a write-read request sequence as an example. A user writes a record into a table (step 1 & 2), and then immediately sends a query to retrieve it. When coordinator receives this query, it sends both the query and versions (denoted as V_1) cached from previous flush responses (for bounded-staleness read, or pulled from write nodes for real-time read, i.e., step 3) to the corresponding read nodes (step 4). For each partition, the read node compares its local version (denoted as V_2) with V_1. If V_1 is not larger than V_2, the node can directly execute the query. Otherwise, it has to pull the latest data from write nodes (step 5) and updates its local copy first.

Following above actions, we can ensure data visibility between read nodes and write nodes for real-time queries. However, if read nodes issue pull requests to write nodes and wait for the required data, the latency would be prohibitively high. We optimize this by replacing read node pulls with write node pushes. When write nodes observe newly written data, they push it along with the versions to corresponding read nodes actively.
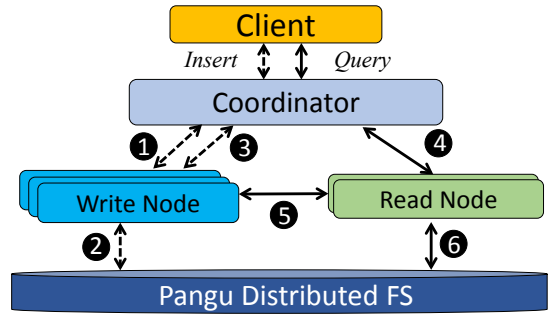

Figure 5: Real-time read workflow.

3.4.3 Reliability and Scalability

Reliability. AnalyticDB provides high reliability for both write nodes and read nodes. For write nodes, when a worker fails, the master will evenly distribute partitions that worker serves to other available write nodes. When the master fails, a new master will be elected from active workers.

For read nodes, users can specify a replication factor (e.g., 2 by default), and different replicas of the same node are deployed on different physical machines (detailed in Section 3.5). When a read node fails while serving a query, the coordinator will re-send that query to other replicas automatically, which is transparent to users. Note that read nodes will not be blocked by failures of write nodes when pulling new data from them. If read nodes fail to contact a write node, they can directly read data from Pangu (though resulting in higher latency) and continue their executions (step 6 in Figure 5).

Scalability. AnalyticDB also guarantees high scalability for write nodes and read nodes. When a new write node is added, the master will adjust table partition placement in order to ensure load balance. The new placement is updated to ZooKeeper, and coordinators can issue subsequent write requests based on the new information. The scalability of read nodes works in a similar way, except that table partition placement is adjusted by coordinators.

3.5 Cluster Management

The cluster management for AnalyticDB is designed for multi-tenant purposes. That is to say, there could be many AnalyticDB instances existing within one cluster. We design and implement a cluster management component called Gallardo, which utilizes Control Group technology to isolate resources (CPU cores, memory, network bandwidth) among different AnalyticDB instances and guarantee their stability. When a new AnalyticDB is created, Gallardo allocates required resources for it. During allocation, Gallardo carefully places different roles (Coordinator, Write Node, Read Node) and replicas of Read Node into different physical machines, so as to obey the reliability requirement. Note that Gallardo has no confliction with Fuxi. Gallardo is responsible for allocating and isolating resources among different AnalyticDB instances, while Fuxi utilizes available resources from all AnalyticDB instances for computation tasks.

4. STORAGE

The storage model of AnalyticDB supports structured data and other complex data types, such as JSON and vectors. We first discuss our hybrid row-column storage layout, followed by its fast and powerful index engine.
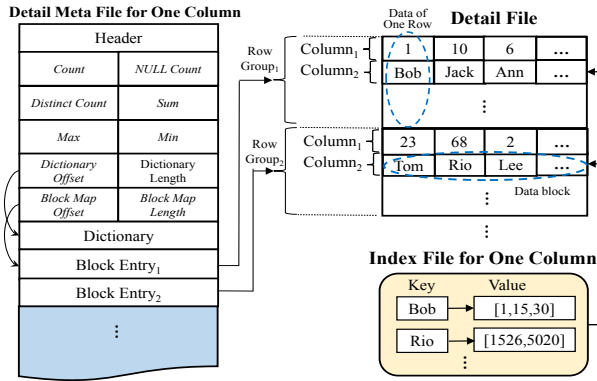
Figure 6: Data format in hybrid row-column store, including meta data and index format.

4.1 Physical Data Layout

This section first presents the data layout and meta data in AnalyticDB, and then explains how data is manipulated.

4.1.1 Hybrid Row-Column Store

One design goal of AnalyticDB is to support both OLAP-style and point-lookup queries. OLAP-style queries often involve a small set of columns in a wide table. Column-store [34] is well-suited for these queries due to its effective data compression and I/O reduction, but struggles for point-lookup queries that need to access one or several entire rows. Row-store outperforms in point-lookup cases, however, it amplifies access cost for OLAP-style queries [12].

To address this dilemma, we propose a hybrid row-column storage layout, as shown in Figure 6. In our design, data in each table partition is maintained in a single file (called *detail file*), which is divided into multiple *row groups*. Each row group contains a fixed number[1] of rows. Within a row group, all values from the same column are contiguously grouped as a data block, and all data blocks are stored in sequence. The data block is the basic operational unit (e.g., for fetch and cache) in AnalyticDB, and it helps achieve high compression ratio to save storage space. Such a hybrid design is able to balance OLAP-style and point-lookup queries with acceptable overhead [12, 20, 34]. Similar to column-store, hybrid-store still clusters data by columns, benefiting OLAP-style queries. Though an entire column resides in multiple data blocks within different row groups, only a small number of sequential seeks are required to fetch all data. As we observe in our real-world AnalyticDB service, this overhead contributes to less than 5% of the overall query latency. For point-lookup queries, it also preserves good performance, as all columns of a specific row are stored in the same row group. Row assembling only involves short-distance sequential seeks [23], instead of cross-segment seeks in column-store.

Complex-typed Data. The hybrid row-column store suits for short columns, e.g., numeric and short string types, but is not for complex-typed data (e.g., JSON and vectors), as this data is of variable sizes and usually much larger. Grouping these rows into fixed-count row groups could result in unexpectedly huge blocks. To address this issue, a fixed-size storage model is designed for complex-typed data. It leverages another level of blocks, namely FBlock, each

---

[1]This number is configurable, set as 30,000 by default according to our production practice.
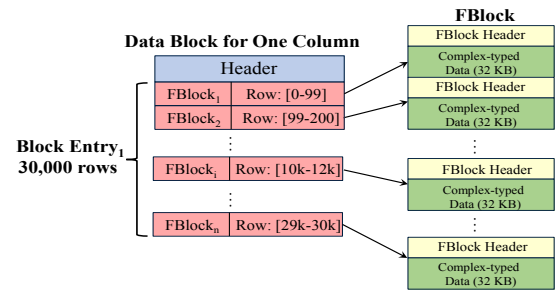
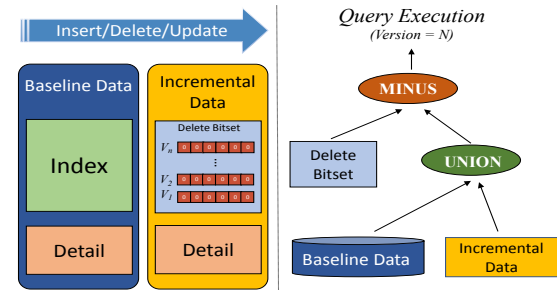

Figure 7: The complex-typed data format.

Figure 8: Manipulation and query execution over storage.

with a fixed size of 32KB. In particular, a data block (with 30,000 rows) further distributes its rows into multiple F-Blocks and stores pointers to these FBlocks instead. In this manner, the data block is still with fixed row count, and all FBlocks are stored in a separate file, as shown in Figure 7. However, the number of rows contained in a FBlock can vary from less than one (i.e., a partial row) to many. To support fast search, we maintain in the data block a block entry for each FBlock, as shown in the left part of Figure 7. Each entry contains two identifiers, i.e., start row and end row of the corresponding FBlock. A row can be split into multiple contiguous FBlocks. For example in Figure 7, FBlock1 and FBlock2 stores rows [0, 99] and [99, 200] respectively, indicating that row 99 is divided into two FBlocks. To access it, we first scan block entries from the data block to locate involved FBlocks (e.g., FBlock1 and FBlock2), and then fetch and concatenate partial rows.

4.1.2 Meta Data

Each column in the detail file has its own meta data that accelerates retrieval over huge amounts of data in that column. This meta data is stored as a separate file for each column, called *detail meta file* as illustrated in Figure 6. It is small in size (i.e., below 1MB) and is cached in memory for frequent access. Meta data of each column consists of four sections: a *header* that contains a version number and the total length of this detail meta file; a *summary* that contains important statistics required by query optimizer (Section 5.1), such as number of rows, number of NULLs, value cardinality, sum (SUM) and maximum/minimum (MAX/MIN); a *dictionary* that is automatically enabled for columns with low cardinality for space saving; and a *block map* that keeps one entry for each data block, containing its offset/length in the detail file for fast access.

4.1.3 Data Manipulation

AnalyticDB uses Lamda architecture for underlying storage (see Figure 8), which contains baseline data and incre-

---

Algorithm 1: INSERT(SQL, version)

Input: SQL statement and version number
```
/* parse multiple column values from SQL   */
```
values = parse(SQL);
```
/* append values to tail of incremental data   */
```
row_id = incremental_data.append(values);
```
/* add a new bit to the delete bitset   */
```
delete_bitset[row_id] = 0;
```
/* create a snapshot of delete_bitset   */
```
delete_bitset_snap = create_snapshot(delete_bitset);
```
/* put a version-snapshot pair into the map   */
```
snap_map.put(version, delete_bitset_snap);

Algorithm 2: DELETE(SQL, version)

Input: SQL statement and version number
```
/* search according to WHERE conditions   */
```
row_ids = search(baseline_data, incremental_data, SQL.where);
```
/* delete records satisfying conditions   */
```
for *each row_id in row_ids* **do**
  delete_bitset[row_id] = 1;

delete_bitset_snap = create_snapshot(delete_bitset);
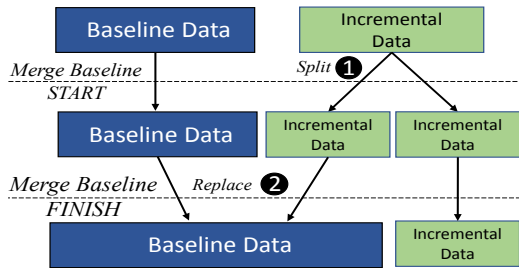snap_map.put(version, delete_bitset_snap);

Figure 9: The process of merging baseline data and incremental data.

mental data. The baseline data stores historical data including both index and row-column data. The incremental data keeps the newly written data, and does not contain full-index but a simple sorted index (detailed in Section 4.2.5). Incremental data occurs only on read nodes when they pull and replay logs from write nodes. Baseline and incremental data follow the identical data and meta formats above.

Query Execution. In order to support UPDATE[2] we use bit-set to record the row ids of deleted data. Copy-on-write technology is used to support MVCC (Multi-version concurrency control) [15]. When a row is updated or deleted, a bit-set snapshot along with a version number is stored in an in-memory map for serving subsequent queries. This delete bit-set is divided into small compressed segments, so that snapshots can share unchanged segments to make them space-efficient. Furthermore, when a new version of the snapshot is created, the oldest version will be eliminated once there is no query running on it. Algorithms 1, 2 and 3 explain how INSERT, DELETE and FILTER queries are executed on both baseline and incremental data, following steps in Figure 8. To execute a query, a version number is given firstly, and then corresponding delete bit-set snapshots of

---

[2]Currently, we only support primary-key-based update to avoid changing too much data in one operation. An update is treated as the combination of a delete and an insert.

Algorithm 3: FILTER(conditions, version)

Input: filter conditions and version number
Output: row ids satisfying conditions
```
/* get delete bitset according to version   */
```
delete_bitset_snap = snap_map.get(version);
```
/* get row-ids by searching index with conditions   */
```
row_ids = search(baseline_data, incremental_data, conditions);
```
/* remove deleted row-ids from the results   */
```
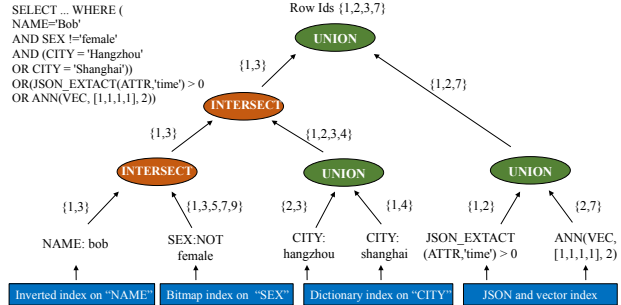return minus(row_ids, delete_bitset_snap);

Figure 10: Query over indexes on all columns.

both baseline and incremental data are referenced. Qualified row-ids are obtained from full-index (on baseline data) and sorted index (on incremental data) respectively. After that, we filter out deleted rows from referenced delete bitsets to get the final results.

Merge Baseline Data with Incremental Data. As new data is continuously written, searching on incremental data is significantly slowed down. Therefore, a build process is started asynchronously to merge incremental data into baseline data. During this process, the deleted records will be ignored and a new index will be created correspondingly. As illustrated in Figure 9, the merge procedure is as follows: When the build process starts, we make current incremental data immutable and create another incremental data instance to handle new arrivals. Before the build process finishes, all queries are executed on the baseline data, stale incremental data and new incremental data. Once the new version of baseline data is merged, the old baseline data and stale incremental data can be safely removed. At this time, the new baseline data, along with new incremental data, serves subsequent queries.

4.2 Index Management

Indexes are a key component in almost all databases to improve query performance. However, existing indexing approaches are not able to fully satisfy requirements in OLAP applications. For example, B^+-trees are costly to update due to node splitting, and hence are only affordable on carefully selected columns. Systems like Druid [36] choose bitmap-based inverted indexes to build on more columns, but only suitable for some specific data types (e.g., String). Recently, with the increasing need to query on complex-typed data (e.g., JSON, vector and texts), indexes on these types should also be supported. Moreover, most systems build indexes in the write path [8, 6, 5, 10], which significantly limits write performance.

We therefore design and implement an index engine that builds indexes for structured and complex-typed data without affecting write throughput. This engine builds indexes

on all columns to fully support ad-hoc queries, and completely removes index construction from the write path. Several sophisticated designs are made to minimize the storage overhead and maximize the performance.

4.2.1 Index Filtering

Each column in a partition is built with an inverted index, which is stored in a separate file. In the inverted index, the index key is the value from the original column, while the value is the list of corresponding row ids (i.e., row numbers). According to Section 4.1.1, we can easily locate a row via its id, since each row group has a fixed number of rows.

With indexes on all columns, AnalyticDB is able to support high-performance ad-hoc queries. Figure 10 gives a SQL filtering example that contains conditions on both structured data and complex-typed data. For each condition, the index engine filters on its corresponding index and obtains partial results, i.e., a set of qualified row ids. After that, all partial results are merged into the final results through intersection, union, minus, etc. To merge these results, 2-way merging is commonly used in most databases, but it incurs huge memory usage and suffers low concurrency. To mitigate this impact, we apply K-way merging [17] instead, which ensures sub-second query latency on large datasets.

Index Path Selection. However, the overuse of index filtering on all columns may sometimes deteriorate query performance. For example, for two conditions A and B, if the partial results of A is much smaller than that of B, it is more cost-effective to obtain results of A first and further filter them with B, rather than getting both partial results and merging them. To address this issue, we propose a runtime filter-ratio-based index path selection mechanism, which evaluates the *filter ratio* of each condition to decide whether to use the corresponding index at runtime. The filter ratio is defined as the number of qualified rows (retrieved from index) divided by the total number of rows (retrieved from meta data). AnalyticDB uses indexes to filter conditions in ascending order based on their filter ratios. After one condition filtering is finished, if the joint filter ratio of all processed conditions (i.e., calculated by ratio multiplication) is small enough (e.g., less than one percent of total rows), this process termrninates and all previously obtained partial results are K-way merged. Subsequent conditions are directly filtered on these row ids instead of on indexes.

4.2.2 Index for Complex-Typed Data

JSON. When a JSON object is inserted, we flatten hierarchical JSON attributes into multiple columns and then build inverted index for each column. For example, given a JSON object {`id`, `product_name`, `properties` {`color`, `size`}}, it is flattened into columns `id`, `product_name`, `product_properties.color` and `product_properties.size`, each of which is built an index. We apply PForDelta algorithm [39] to compress the row-ids under each index key. Furthermore, a JSON object may contain thousands of attributes (i.e., thousands of indexes). We pack all indexes of one object in a single file to limit the number of files. With the index, AnalyticDB can directly fetch the object according to the predicates in JSON format, which is more efficient than reading and parsing blocks of JSON data from disk.

Full-text. For full-text data, AnalyticDB extends the inverted indexes by storing more information, including term frequency and mapping from document to terms. We then use the popular TF(Term Frequency)/IDF(Inverse Docu-

Table 1: Comparison between AnalyticDB (ADB) and Greenplum (GP) on building all-column index of 1TB data.

	ADB	GP
Index Space	0.66TB	2.71TB
Index Building Time	1 hour	0.5 hour
Asynchronous?	Yes	No
Data Insertion Time	4,015s	20,910s

ment Frequency) score to calculate the similarity between the query and texts in the database. Only those objects with scores above a threshold are returned to users.

Vector Data. Feature vectors are a common component of many computer vision tasks, such as object/scene recognition and machine learning, where a high-dimensional vector can be extracted from an image via a trained AI model. The similarity of two objects can be measured by the distance of their feature vectors. In queries for vector data, users always require nearest neighbour search (NNS), aiming to find objects that are the closest to the query point in the database. Formally, NNS can be defined as finding the object $NN(q)$ in a finite set $\mathcal{Y} \subset \mathbb{R}^D$ of vectors stored in the same column in the database, who minimizes the distance to the query vector $q \in \mathbb{R}^D$, as Equation 1. AnalyticDB supports different similarity metrics d like Euclidean distance and Cosine distance, which can be specified in SQL language.

$$NN(q) = \arg\min_{y \in \mathcal{Y}} d(q, y) \tag{1}$$

The most brute-force way for NNS is to scan all vectors linearly in the database, compute the distances with the query vector and finally return the top-k results. To avoid such exhaustive search, we implement and combine Product Quantization (PQ) [26] and Proximity Graph (k-NNG) [22]. PQ and k-NNG are experimentally-proven efficient approximate NNS approaches [4] that fetch NN (or k-NN) with high probability. PQ has a smaller index size by decomposing the vector space, while k-NNG holds better search performance and accuracy through its efficient well-connected graphical index. AnalyticDB adaptively chooses the most appropriate one for vector data, depending on the memory resources as well as accuracy and efficiency requirements of users.

4.2.3 Index Space Saving

We apply an adaptive method to reduce index sizes. For each key-value pair in the index, AnalyticDB automatically chooses bitmap or integer array to hold the value based on their space consumption. For example, if the index value is [1,2,8,12], bitmap (2 bytes) is cheaper than integer array (4 bytes). But if the index value is [1,12,35,67], integer array (4 bytes) is a better choice than bitmap (9 bytes). By adopting this method, the overall index size can be reduced by 50%. We also allow users to disable indexes on specific columns to trade latency for space.

4.2.4 Asynchronous Index Building

AnalyticDB serves tens of millions of write requests per second, and thus is not affordable to build all-column indexes in the write path. Alternatively, the index engine builds index asynchronously. Recall that in AnalyticDB the write path ends when write nodes flush write logs to Pangu (Section 3.4.1). The index engine periodically builds inverted indexes on these new writes (i.e., incremental data), and merges them with existing full indexes at background.
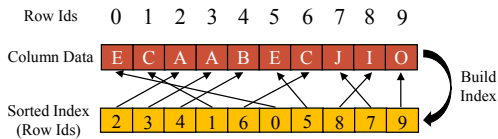
Figure 11: Sorted Index for Incremental Data

Such asynchronous approach completely hides the overhead of building from users, and preserves both query efficiency and write throughput. The process for building and merging indexes is translated into many Map-Reduce tasks. With the help of Fuxi [38], these tasks run concurrently and automatically during off-peak periods within AnalyticDB cluster, introducing acceptable overhead.

Table 1 compares building all-column index for 1TB data on AnalyticDB and Greenplum [5], a column-store-based OLAP database system. We see that AnalyticDB only uses 0.66TB extra space for indexes, which is much smaller than 2.71TB in Greenplum. Although AnalyticDB doubles the index building time, this asynchronous process does not impact performance of online reads and writes. As shown in Table 1, the time for real-time ingestion (INSERT) of 1TB data in Greenplum is about four times longer than in AnalyticDB. Thus, AnalyticDB yields acceptable overhead in exchange for significant performance improvement for ad-hoc queries, which is evaluated in Section 6.

4.2.5 Index For Incremental Data

The adoption of asynchronous indexes brings in a performance gap: before the new index is online, incremental data lack the support of indexes, and thus they need be scanned to serve a query with high latency. To close this gap, the index engine independently builds *sorted indexes* in read nodes for incremental data. The sorted index is actually an array of row ids in the data block. As illustrated in Figure 11, for an ascending sorted index, the i-th element T_i represents that the i-th smallest value in the data block is at row T_i. A search in incremental data is then converted to a binary search, reducing the complexity from $O(n)$ to $O(\log n)$. To store the sorted index, we allocate an additional header in each data block. As there are 30k rows in a block and each row id is a short integer, the size of the header, i.e., the sorted index, is just about 60KB. Before flushing a data block, the index engine builds the sorted index and dumps it to the head of the file. This building process is executed locally in read nodes and is quite lightweight.

4.2.6 Index Cache for Conditions

Traditional databases cache indexes (at the granularity of index page) in the memory to reduce costly disk I/Os. AnalyticDB applies not only an index-page cache, but a more aggressive query-condition cache. This query-condition cache treats query conditions (e.g. `id < 123`) as keys, and query results (i.e., row ids) as values. In consequence, repeated filtering over index pages can be completely avoided. When the query-condition cache misses, we can access indexes in index-page cache to compute query results.

One challenge in our two-level cache policy is that user conditions change continuously and dramatically, resulting in frequent cache eviction. However, we observe that this will not affect overall cache effectiveness much: 1) the conditions with large-size results are rare and do not change frequently (e.g., `WHERE city='Beijing'`), so that their caches can last for a long time; 2) the conditions with small-size
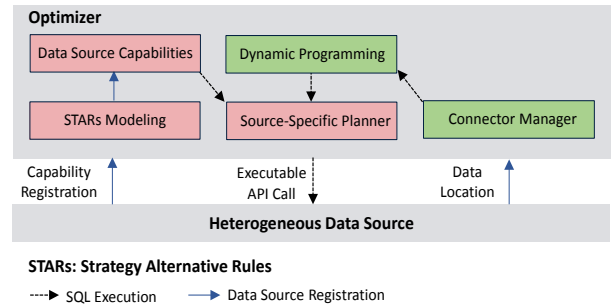

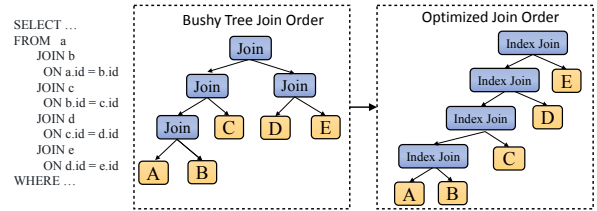**Figure 12: The STARs framework of AnalyticDB.**

Figure 13: Storage-aware join optimization.

results are massive and change dramatically (e.g., `WHERE user_id=XXX`), but their results can be re-computed with low cost. In summary, costly computations are usually well cached to save resources, while light queries are re-computed without much extra overhead. This observation ensures the effectiveness of our index cache.

5. OPTIMIZER AND EXECUTION ENGINE

In this section, we discuss the novel optimizations adopted by optimizer and execution engine respectively, which further improve query latency and concurrency.

5.1 Optimizer

The AnalyticDB optimizer provides both CBO (cost-based optimization) and RBO (rule-based optimization), and targets the real-time online analytics that require extremely low response time and high concurrency. It contains a rich set of relational algebra conversion rules to ensure that the optimal plan can be always selected. These rules include: basic optimization rules (e.g., cropping, pushdown/merge, deduplication, constant folding/predicate derivation); probe optimization rules for different Joins (e.g., BroadcastHashJoin, RedistributedHashJoin, NestLoopIndexJoin), Aggregate, Join-Reorder, GroupBy pushdown, Exchange pushdown, Sort pushdown, etc.; and advanced optimization rules (e.g., Common Table Expression). Other than the generic CBO/RBO above, two key features are developed, i.e., storage-aware optimization and efficient real-time sampling.

5.1.1 Storage-Aware Plan Optimization

Predicate Push-down. Predicate (i.e., condition) push-down is to extract the relational algebraic calculations in SQL that can take advantage of the underlying storage, and to convert this query plan into two equivalent parts (i.e., for the compute layer and the storage layer respectively). Since there is no clear boundary in the original query plan to support this separation, it completely relies on the optimizer. Predicate push-down has already been implemented in many distributed databases, but mainly focusing on AND operators of single-column conditions. They do not consider

other common operators, such as functions and joins, which are usually implemented in the compute layer. This is because many existing databases do not provide the interface for the storage layer to register its advanced capability. As a result, the storage layer can only do a single column or a combination of condition filtering.

AnalyticDB introduces a STARs (STrategy Alternative Rules) [30, 14] framework to make the optimizer extensible for predicate push-down, as illustrated by Figure 12. STARs provide a high-level, declarative, implementation-independent specification of legal strategies for query execution. Each STAR defines a set of high-level constructs from low-level database operators or other STARs. With the STARs framework, AnalyticDB abstracts the capability of heterogeneous data sources according to the dimensions of relational algebra, and characterizes the capability of storage as relational algebra that can be utilized. In addition, the STARs framework also provides cost calculation. That is, performing a push-down is not simply dependent on the capability of storage, but also on the cost of its relational algebra ability. During the dynamic planning, both the cost and execution capability are used as reference factors at the same time, so as to avoid blindly pushing down and causing performance deterioration. This is important in low latency and high concurrency environment. After the optimizer completes the initial distributed execution plan, the relational algebraic operators applicable for target data sources are encapsulated by means of dynamic programming, and transformed into corresponding storage API calls.

Join Push-down. Data redistribution is another important aspect of distributed database execution plan. It is different from those in traditional databases, mainly due to the mismatch between the physical distribution characteristics of data and the logical semantics of relational algebra. For example, in SQL statement "`SELECT T.tid, count(*) FROM T JOIN S ON T.sid = S.sid GROUP BY T.tid`", based on whether table T and S are hashed by the same field and with partitions placed at the same read node (Section 3.4.2), AnalyticDB is able to select the best join push down strategy. The avoidance of data redistribution is very important, since the cost of physical data redistribution is prohibitive, which involves serialization, de-serialization, network overhead, etc. In the case that table T and S are not hashed by the same field, AnalyticDB clearly knows shuffling which table is more efficient, by obtaining the sizes of T and S from the underlying storage. As mentioned previously, the optimizer expands and calculates the cost of all possible execution plans. It is in this way that AnalyticDB achieves an execution plan that is optimal for data characteristics at different data sizes.

Index-based Join and Aggregation. Index-on-all-columns further eliminate the overhead on building hash indexes, by doing lookup on existing indexes instead. When adjusting the order of join, the optimizer avoids generating BushyTree and prefers LeftDeepTree, when most of join columns are partition columns and have indexes. With the LeftDeepTree, AnalyticDB utilizes existing indexes better (see Figure 13). Additionally, we also push down predicates and aggregations. For example, aggregations like `count` can be returned directly from indexes; and filtering can be evaluated solely on indexes. All these optimizations reduce the query latency while improving the cluster utilization, which enables AnalyticDB to support high concurrency easily.

5.1.2   Efficient Real-time Sampling

Cost estimation is the foundation for cost-based optimization and is determined by the cardinality estimation that heavily relies on the available statistics. In modern databases, statistics are collected and utilized in a limited way where data skewness and correlation are not well handled, resulting in sub-optimal query plans. Besides, since one of our system design goals is the short response time of queries that are either simple or complex, conventional approaches (such as real-time statistics, predicate selectivity profiling, and execution results feedback) are impractical due to their overheads and complexities. Instead, we design and implement an efficient sampling-based cardinality estimation framework. Our framework takes advantages of the AnalyticDB's high-performance storage engine that provides efficient data access and evaluation through abundant types of indexes, caches, and optimized computation. At optimization time, the optimizer sends request on the sampling predicates (individual or compound decided by optimization rules) to the storage engine through the framework APIs. Storage engine then accesses the sampled data via appropriate indexes/caches, evaluates the predicates via optimized computation paths, and returns the cardinality results. The optimizer utilizes the sampled cardinality to estimate candidate plans and choose the optimal one.

Although our sampling framework estimates the cardinality efficiently, further optimizations are carried out to further reduce the overhead, especially for those critical business scenarios where queries run in sub-seconds. Such optimizations include caching previously sampled results (and cardinality estimates), optimized sampling algorithm, and improved derived cardinality, etc. By applying all these, our sampling-based cardinality framework is able to estimate the cardinality at extremely low overhead in milliseconds while providing high estimation accuracy.

5.2   Execution Engine

AnalyticDB provides a general-purpose, pipeline-mode execution engine, as well as a DAG [27] (Directed Acyclic Graph) running framework on top of this engine. It is suitable for both small (with low latency) and large-scale (with high throughput) workloads. AnalyticDB execution engine is column-oriented, which takes advantages of the underlying hybrid store clustering data on columns. Compared to row-oriented execution engine, this vectorized engine is cache-friendly and avoids loading unnecessary data into memory.

Like many OLAP systems, runtime code generator [32] (CodeGen) is used to improve the efficiency of CPU-intensive operations. AnalyticDB CodeGen is based on ANTLR ASM [2] that dynamically generates code for the expression trees. This CodeGen engine also takes runtime factors into consideration, allowing us to leverage the power of heterogeneous hardware in a task-level granularity. For example, most types of data (such as `int` and `double`) in vectorized engine are aligned. In a heterogeneous cluster with CPUs supporting AVX-512 instruction set, we can generate byte-codes using SIMD instructions to improve performance. In addition, by consolidating the internal data representation between storage layer and execution engine, AnalyticDB is able to operate directly on serialized binary data rather than Java objects. This helps eliminate the overhead of serialization and de-serialization, which accounts for more than 20% of time when shuffling a large amount of data.

Table 2: Three kinds of queries for evaluation.

Query type	Query
Full Scan (Q1)	`SELECT * FROM orders ORDER BY o_trade_time LIMIT 10`

Point Lookup (Q2)	`SELECT * FROM orders`
	`WHERE o_trade_time BETWEEN '2018-11-13 15:15:21' AND '2018-11-13 16:15:21'`
	`AND o_trade_prize BETWEEN 50 AND 60 AND o_seller_id=9999 LIMIT 1000`

Multi-table Join (Q3)	`SELECT o_seller_id, SUM(o_trade_prize) AS c FROM orders`
	`JOIN user ON orders.o_user_id = user.u_id`
	`WHERE u_age=10 AND o_trade_time BETWEEN '2018-11-13 15:15:21' AND '2018-11-13 16:15:21'`
	`GROUP BY o_seller_id ORDER BY c DESC LIMIT 10;`

6. EVALUATION

In this section, we evaluate AnalyticDB in both real workloads and TPC-H benchmark [11] to demonstrate AnalyticDB's performance under different types of queries and its write capability.

6.1 Experimental Setup

The experiments are conducted on a cluster of eight physical machines, each with an Intel Xeon Platinum 8163 CPU (@2.50GHz), 300GB memory and 3TB SSD. All these machines are connected through 10Gbps Ethernet network. We create one AnalyticDB instance with 4 coordinators, 4 write nodes, and 32 read nodes within this cluster.

Real Workloads. We use two real tables in our production for the evaluation. The first table is called Users Table, which uses `user_id` as its primary key, and has 64 primary partitions (no secondary partition). The second table is called Orders Table, which uses `order_id` as its primary key, and has 64 primary partitions and 10 secondary partitions. These two tables are associated with each other through `user_id`. We use three types of queries generated from our users (as shown in Table 2), ranging from scan, point lookup to multi-table join. Note that all three queries contain `o_trade_time`, which is a timestamp column. It is because Druid MUST have a timestamp column as the partition key, and queries without specifying the timestamp column are much slower [36].

Systems to compare against. We compare AnalyticDB with four OLAP systems: PrestoDB [9], Spark-SQL [13], Druid [36], and Greenplum [5]. Greenplum has index on all columns; Druid does not support index on numeric columns; PrestoDB and Spark-SQL keep data in Apache ORC (Optimized Record Columnar) File [3] and do not have index on any column. All systems are run in their default configurations. Note that Druid does not support complex queries like JOIN, so that most of TPC-H queries and Q3 in Table 2 can not be executed. Hence, we omit it in corresponding experiments. In all experiments, the term *concurrency number* refers to the number of concurrently running queries.

6.2 Real Workloads

This section first presents query performance on 1TB data and 10TB data, and then shows the write throughput.

6.2.1 Query on 1TB Data

We generate a dataset of 1TB to run the three queries in Table 2. Figure 14 and Figure 15 show the 50-percentile and 95-percentile query latency of AnalyticDB, PrestoDB, Druid, Spark-SQL, and Greenplum respectively. As can be seen, AnalyticDB achieves a lower latency by at least an order of magnitude compared to other systems.

Q1. With the help of the index engine, AnalyticDB avoids expensive scan and sort over the entire table, which is different from PrestoDB and Spark-SQL. In particular, AnalyticDB distributes the operators of `ORDER BY` and `LIMIT` to each secondary partition, which holds the index for column `o_trade_time`. Since the index is ordered, each partition just traverses the index to obtain qualified row ids, involving only dozens of index entries. Although Greenplum also builds index on all columns, it fails to utilize them for `ORDER BY` operators and executes a full scan, thus is much slower than AnalyticDB. Druid uses `o_trade_time` as the column for range partitioning [36]. When doing `ORDER BY` on this column, Druid filters from the largest range partition. It achieves better performance than Greenplum, but is still slower than AnalyticDB as it still scans all rows in that partition.

Q2. In our dataset, the number of rows satisfying conditions on `o_trade_time`, `o_trade_prize` and `o_seller_id` are 306,340,963, 209,994,127, and 210,408 respectively. Without index support, PrestoDB and Spark-SQL has to scan all rows for filtering. Druid and Greenplum achieve better performance as they can benefit from fast searches on indexed columns. However, Druid only builds indexes on string columns. Greenplum's indexes are available for all columns, but it has to filter multiple conditions sequentially, and does not have cache for unchanged conditions. Compared to them, AnalyticDB directly scans indexes on three columns in parallel, and caches qualified row ids respectively (Section 4.2.6). Hence, the subsequent queries with same conditions could benefit from the index cache.

Q3. As shown in Figure 14 and Figure 15, the 50/95-percentile latency of Q3 is higher than that of Q1 and Q2 under different concurrency levels. This is because Q3 is a much more complicated query, i.e., a multi-table join scan combined with `GROUP BY` and `ORDER BY` operators. Though the latency is slightly high due to the query complexity, AnalyticDB still ensures that the optimal execution can be achieved. In particular, AnalyticDB translates the join operator into equality sub-queries and makes use of indexes to complete these sub-queries. It further leverages indexes to execute `GROUP BY` and `ORDER BY` operators and avoids the overhead of building a hashmap. Greenplum is slower than AnalyticDB, because it bears the hashmap overhead from hash join. To make it fair, we also evaluate AnalyticDB in hash join mode and are able to achieve comparable performance with Greenplum.

6.2.2 Query on 10TB Data

We further generate a larger dataset of 10TB data and increase the level of concurrency. As the comparison systems are much slower than AnalyticDB in large datasets and higher concurrency, we omit them from this analysis.
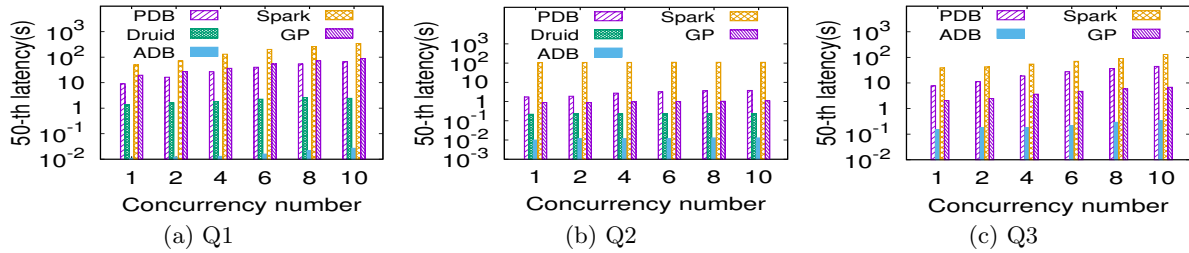
(a) Q1   (b) Q2   (c) Q3

Figure 14: 50-percentile latency, 1TB data (PDB for PrestoDB, ADB for AnalyticDB, GP for Greenplum).
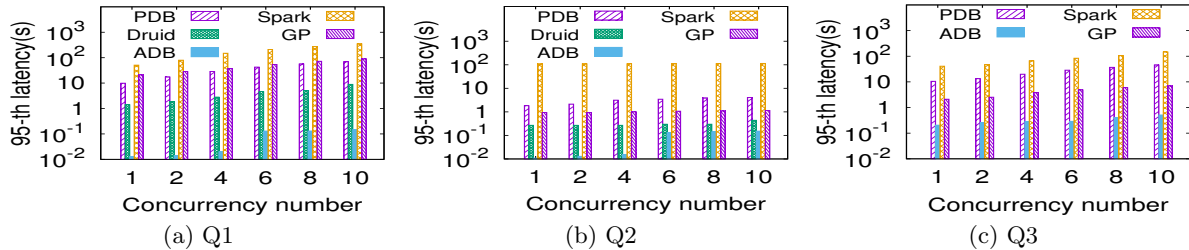

(a) Q1   (b) Q2   (c) Q3

Figure 15: 95-percentile latency, 1TB data (PDB for PrestoDB, ADB for AnalyticDB, GP for Greenplum).
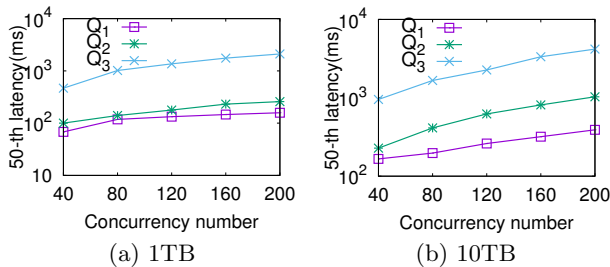

(a) 1TB   (b) 10TB

Figure 16: 50-percentile latency of AnalyticDB on 1TB and 10TB data.

Figure 16 illustrates the 50-percentile latency of three queries on both 1TB and 10TB data. We can see that, for Q1 and Q2, the latency is within hundreds of milliseconds under different concurrency levels. For Q3, the latency with 200 concurrency is much higher than that with 40 concurrency. The reason is that the computing capacity under 8 machines has been saturated. Specifically, with 64 primary and 10 secondary partitions, the real concurrent threads could reach 128,000 under 200 concurrency. There are in total $48 \times 8 = 384$ CPU cores on these eight machines. As Q3 is computing-intensive, the performance is deteriorated due to frequent context switch, suffering from high latency.

From Figure 16 we can see that the variation trend on 10TB data under different concurrency levels is similar to that on 1TB data. That is, the performance is not affected dramatically in spite of the increase in data size. The query latency on 10TB just doubles that on 1TB, because AnalyticDB searches indexes for row ids first and only needs to fetch qualified rows. With the help of index cache, index lookup is cost-effective and brings down the overall overhead. In summary, the performance of AnalyticDB is slightly impacted by the table size, but is more dominated by the computation on indexes as well as the number of qualified rows.

6.2.3 Write Throughput

To evaluate the write performance of AnalyticDB, we insert records, each with 500 bytes, into Orders Table. Table 3 illustrates the write throughput (write requests per second). Thanks to the read/write decoupling architecture and asyn-

Table 3: Write throughput under different numbers of write nodes.

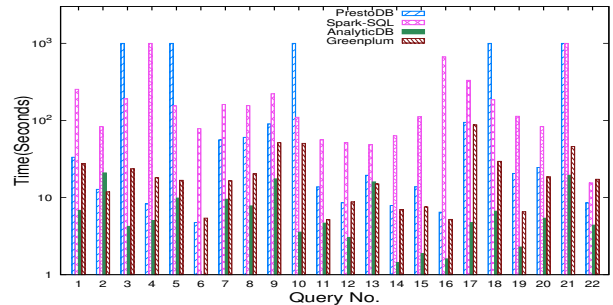| write node number	2	4	6	8	10
write throughput	130k	250k	381k	498k	625k

Figure 17: TPC-H performance comparison.

chronous index building, the write throughput grows nearly linearly with the increasing number of write nodes, until Pangu is saturated. When the number of write nodes reaches 10, the write throughput is 625,000 and the bandwidth corresponds to about 300 MB/s. Note that index building tasks are distributed over the entire AnalyticDB cluster with affordable overhead (as evaluated in Table 1 in Section 4.2.4), which will not affect both query efficiency and write throughput.

6.3 TPC-H Benchmark

We generate 1TB data for TPC-H evaluation. Figure 17 illustrates the performance comparison among AnalyticD-B, PrestoDB, Spark-SQL, and Greenplum [3]. AnalyticDB achieves the smallest running time for 20 out of 22 queries, and outperforms the second best, i.e., Greenplum, by 2x. Compared to Spark-SQL, AnalyticDB adopts pipeline process model and indexes, which is faster than stage-based process. PrestoDB also adopts pipeline process, but it lacks indexes on columns. Although Greenplum also has both

---

[3]In Figure 17, if the time of a query is 1000s, it means the system encountered an exception when running the query and we did not get a result, e.g., query No.3 of PrestoDB and query No.21 of Spark-SQL.

pipeline process and all-column index, AnalyticDB preserves four additional advantages. First, AnalyticDB uses hybrid row-column storage, while Greenplum uses column-store. About half of columns are involved in a common TPC-H query, and hence AnalyticDB can fetch multiple columns of a row with a single I/O. Second, AnalyticDB's runtime cost-based index path selection (using real intermediate results) leads to better table access plans, compared to statistics-based planning in Greenplum. Third, AnalyticDB combines K-ways merging with composite predicates pushdown. Fourth, AnalyticDB incorporates vectorized execution engine and applies optimized CodeGen to all operators and expressions. For query No.2, AnalyticDB is slower than PrestoDB and Greenplum, because it chooses a different join order for multi-table join.

7. CONCLUSION

This paper presents AnalyticDB, a high-concurrent, low-latency, and real-time OLAP database at Alibaba. AnalyticDB has an efficient index engine to build index for all columns asynchronously, which helps improve query performance and hide index building overhead. With careful designs, the all-column index just consumes 66% more storage. AnalyticDB extends hybrid row-column layout to support both structured and other complex-typed data that may be involved in complex queries. To provide both high-throughput write and high-concurrent query, AnalyticDB follows a read/write decoupling architecture. Moreover, we enhance the optimizer and execution engine in AnalyticDB to fully utilize the advantages of our storage and indexes. Our experiments have showed that all these designs help AnalyticDB achieve better performance compared to state-of-art OLAP systems.

8. REFERENCES

[1] Alibaba Cloud. https://www.alibabacloud.com.
[2] ANTLR ASM. https://www.antlr.org.
[3] Apache ORC File. https://orc.apache.org/.
[4] Benchmarking Nearest Neighbours. https://github.com/erikbern/ann-benchmarks.
[5] Greenplum. https://greenplum.org/.
[6] MySQL. https://www.mysql.com/.
[7] Pangu. https://www.alibabacloud.com/blog/pangu—the-high-performance-distributed-file-system-by-alibaba-cloud_594059.
[8] PostgreSQL. https://www.postgresql.org/.
[9] Presto. https://prestodb.io/.
[10] Teradata Database. http://www.teradata.com.
[11] TPC-H Benchmark. http://www.tpc.org/tpch/.
[12] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, pages 967–980. ACM, 2008.
[13] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394. ACM, 2015.

[14] J. Backus. *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*. ACM, 2007.
[15] P. A. Bernstein and N. Goodman. Multiversion concurrency control-theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.
[16] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.
[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.
[18] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[19] A. Eisenberg, J. Melton, K. Kulkarni, J.-E. Michels, and F. Zemke. Sql: 2003 has been published. *ACM SIGMOD Record*, 33(1):119–126, 2004.
[20] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: a main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.
[21] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *SIGMOD*, pages 1917–1923. ACM, 2015.
[22] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI*, pages 1312–1317, 2011.
[23] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498. VLDB Endowment, 2006.
[24] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 8. Boston, MA, USA, 2010.
[25] J.-F. Im, K. Gopalakrishna, S. Subramaniam, M. Shrivastava, A. Tumbde, X. Jiang, J. Dai, S. Lee, N. Pawar, J. Li, et al. Pinot: Realtime olap for 530 million users. In *SIGMOD*, pages 583–594. ACM, 2018.
[26] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011.
[27] F. V. Jensen. *An introduction to Bayesian networks*, volume 210. UCL press London, 1996.
[28] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. Impala: A modern, open-source sql engine for hadoop. In *Cidr*, volume 1, page 9, 2015.
[29] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.
[30] G. M. Lohman. *Grammar-like functional rules for representing query optimization alternatives*, volume 17. ACM, 1988.
[31] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *PVLDB*, 3(1-2):330–339, 2010.
[32] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.
[33] K. Sato. An inside look at google bigquery.(2012). *Retrieved Jan*, 29:2018, 2012.
[34] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *VLDB*, pages 553–564. VLDB Endowment, 2005.
[35] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.
[36] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A real-time analytical data store. In *SIGMOD*, pages 157–168. ACM, 2014.
[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2. USENIX Association, 2012.
[38] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *PVLDB*, 7(13):1393–1404, 2014.
[39] M. Zukowski, S. Heman, N. Nes, and P. Boncz. *Super-scalar RAM-CPU cache compression*. IEEE, 2006.