FusionInsight LibrA: Huawei's Enterprise Cloud Data Analytics Platform

Le Cai[*], Jianjun Chen[†], Jun Chen, Yu Chen, Kuorong Chiang, Marko Dimitrijevic, Yonghua Ding
Yu Dong[*], Ahmad Ghazal, Jacques Hebert, Kamini Jagtiani, Suzhen Lin, Ye Liu, Demai Ni[*]
Chunfeng Pei, Jason Sun, Yongyan Wang[*], Li Zhang[*], Mingyi Zhang, Cheng Zhu
Huawei America Research

ABSTRACT

Huawei FusionInsight LibrA (FI-MPPDB) is a petabyte scale enterprise analytics platform developed by the Huawei database group. It started as a prototype more than five years ago, and is now being used by many enterprise customers over the globe, including some of the world's largest financial institutions. Our product direction and enhancements have been mainly driven by customer requirements in the fast evolving Chinese market.

This paper describes the architecture of FI-MPPDB and some of its major enhancements. In particular, we focus on top four requirements from our customers related to data analytics on the cloud: system availability, auto tuning, query over heterogeneous data models on the cloud, and the ability to utilize powerful modern hardware for good performance. We present our latest advancements in the above areas including online expansion, auto tuning in query optimizer, SQL on HDFS, and intelligent JIT compiled execution. Finally, we present some experimental results to demonstrate the effectiveness of these technologies.

PVLDB Reference Format:
Le Cai, Jianjun Chen, Jun Chen, Yu Chen, Kuorong Chiang, Marko Dimitrijevic, Yonghua Ding, Yu Dong, Ahmad Ghazal, Jacques Hebert, Kamini Jagtiani, Suzhen Lin, Ye Liu, Demai Ni, Chunfeng Pei, Jason Sun, Yongyan Wang, Li Zhang, Mingyi Zhang, Cheng Zhu. FusionInsight LibrA: Huawei's Enterprise Cloud Data Analytics Platform. *PVLDB*, 11 (12): 1822-1834, 2018.
DOI: https://doi.org/10.14778/3229863.3229870

1. INTRODUCTION

Huawei FusionInsight LibrA (FI-MPPDB) [9] is a large scale enterprise data analytics platform developed by Huawei. Previous version known as Huawei FusionInsight MPPDB was launched in 2015 and has been adopted by many customers over the globe, including some of the world's largest financial institutes in China. With the help of the success of Huawei FusionInsight MPPDB, FusionInsight products started appearing in Gartner magic quadrant from 2016.

The system adopts a shared nothing massively parallel processing architecture. It supports petabytes scale data warehouse and runs on hundreds of machines. It was originally adapted from Postgres-XC [14] and supports ANSI SQL 2008 standard. Common features found in enterprise grade MPPDB engine have been added through the years, including hybrid row-column storage, data compression, vectorized execution etc. FI-MPPDB provides high availability through smart replication scheme and can access heterogeneous data sources including HDFS.

Huawei is a leader in network, mobile technology, and enterprise hardware and software products. Part of the Huawei's vision is to provide a full IT technology stack to its enterprise customers that include the data analytics component. This is the main motivation behind developing FI-MPPDB that helps reducing the overall cost of ownership for customers compared to using other DBMS providers. Another advantage is that the full stack strategy gives us more freedom in deciding product direction and quickly providing technologies based on our customer requirements.

The architecture and development of FI-MPPDB started in 2012 and the first prototype came out in early 2014. The main features in our initial system are vectorized execution engine and thread based parallelism. Both features provided significant system performance and were a differentiator for us over Greenplum [31]. The FI-MPPDB release v1 based on the prototype was successfully used by the Huawei distributed storage system group for file meta-data analytics.

With the success of the v1 release, Huawei started to market the FI-MPPDB to its existing customers, especially those in China's financial and telecommunication industry. The product direction and enhancements were driven by our customer requirements, leading to key features in v2 like column store, data compression, and smart workload management. In addition, we developed availability feature to retry failed requests, and for scalability we replaced the original TCP protocol by a new one based on the SCTP protocol.

In 2016, we observed that many of our customers captured a lot of data on HDFS in addition to data on FI-MPPDB. This led us to looking into supporting SQL on Hadoop. We examined competing solutions available that included Apache HAWQ [8], Cloudera Impala [24] and Transwarp Inceptor [17]. We decided to make our data warehouse tightly

[*]The authors, Le Cai, Yu Dong, Demai Ni, Yongyan Wang and Li Zhang, were with Huawei America Research when this work was done.
[†]Dr. Jianjun Chen is the corresponding author, jianjun.chen1@huawei.com

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3229870

integrated with HDFS, allowing our MPP engine to directly work on HDFS data and avoid data movement from HDFS to the FI-MPPDB storage. This approach provides a seamless solution between MPPDB and HDFS with better SQL performance than Transwarp Inceptor, and stronger ACID support than Cloudera Impala. The HDFS support was added to the FI-MPPDB in 2016 and successfully adopted by many of our customers. As a result, FI-MPPDB became part of Huawei FusionInsight product in 2016.

In 2017, we announced our first version of FI-MPPDB on Huawei public cloud, a.k.a. LibrA. Based on our customers' feedbacks on our cloud offering, the top requirements are 1) system availability, 2) auto tuning, 3) support querying large and diversified data models on the cloud, and 4) best utilize modern hardware for achieving high performance over cost ratio.

First, system availability requires that FI-MPPDB should be able to add more nodes (elasticity) or go through an upgrade with minimal impact on customer workloads. This is critical for large systems with petabytes of data that can take hours or even days to migrate and re-balance during system expansion. Similarly, system upgrade can also take hours to finish and make the system unavailable for the end user. These requirements are addressed by our recent features online expansion and online upgrade which greatly minimize the impact of system expansion and software upgrades. Due to the space limitation, we will only cover online expansion in this paper.

Second, DBMS auto tuning minimizes the need for manual tuning by system DBAs. For cloud deployment, such tuning can be complex and costly with the elasticity of the system and the access to heterogeneous data sources. Autonomous database from Oracle [13] emphasizes self managing and auto tuning capabilities, an indication that cloud providers are paying great attention to this area. We have been working on automatic query performance tuning through runtime feedbacks augmented by machine learning.

Third, our cloud customers now have huge amount of data in various formats stored in Huawei cloud storage systems, which are similar to S3 and EBS in AWS. Recently, the notion of Data Lake becomes popular which allows data residing inside cloud storage to be directly queried without the need to move them into data warehouse through ETL. AWS Spectrum [3] and Athena [1] are recent products that provide this functionality. Our product provides SQL on Hadoop (SQLonHadoop) support (a.k.a. ELK in FusionInsight) which was successfully used by many of our customers.

Fourth, modern computer systems have increasingly larger main memory, allowing the working set of modern database management systems to reside in the main memory. With the adoption of fast IO devices such as SSD, slow disk accesses are largely avoided. Therefore, the CPU cost of query execution becomes more critical in modern database systems. The demand from cloud database customers on high performance/cost ratio requires us to fully utilize the great power provided by modern hardware. An attractive approach for fast query processing is just-in-time (JIT) compilation of incoming queries in the database execution engine. By producing query-specific machine code at runtime, the overhead of traditional interpretation can be reduced. The effectiveness of JIT compiled query execution also depends on the trade-off between the cost of JIT compilation and the performance gain from the compiled code. We will introduce our cost based approach in JIT compilation in this paper.

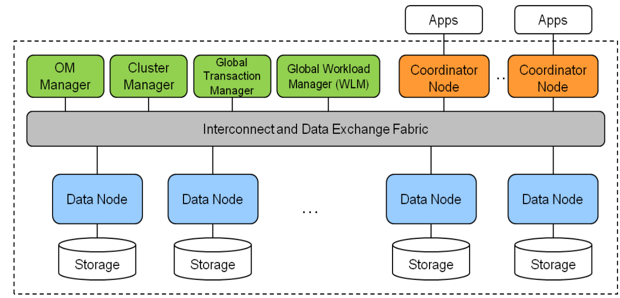

Figure 1: FusionInsight MPPDB System High-level Architecture

The rest of this paper is organized as follows. Section 2 presents an overview of the FI-MPPDB architecture followed by the technical description of the four major directions discussed above. Our experimental results are presented in section 3, which show the efficiency of our online expansion solution, the benefit of auto tuning, the effectiveness of the co-location strategy of SQLonHDFS, and the performance gain from the JIT generated code. Related work is discussed in Section 4 which compares our approaches with other industry leaders in the four cloud key areas. Finally, we conclude our paper in section 5 and discuss future work.

2. TECHNICAL DETAILS

In this section, we first give an overview of FI-MPPDB, and then we present our solutions to the four top customer requirements described in section 1.

2.1 System Overview

FI-MPPDB is designed to scale linearly to hundreds of physical machines and to handle a wide spectrum of interactive analytics. Figure 1 illustrates the high level logical system architecture. Database data are partitioned and stored in many data nodes which fulfill local ACID properties. Cross-partition consistency is maintained by using two phase commit and global transaction management. FI-MPPDB supports both row and columnar storage formats. Our vectorized execution engine is equipped with latest SIMD instructions for fine-grained parallelism. Query planning and execution are optimized for large scale parallel processing across hundreds of servers. They exchange data on-demand from each other and execute the query in parallel.

Our innovative distributed transaction management (GTM-lite) distinguishes transactions accessing data of a single partition from those of multiple partitions. Single-partition transactions get speed-up by avoiding acquiring centralized transaction ID and global snapshot. GTM-lite supports READ COMMITTED isolation level and can scale out FI-MPPDB's throughput manifold for single-partition heavy workloads.

2.1.1 Communication

Using TCP protocol, the data communication requires a huge number of concurrent network connections. The maximum concurrent connections will increase very quickly with larger clusters, higher numbers of concurrent queries, and

more complex queries requiring data exchange. For example, the number of concurrent connections on one physical host can easily go up to the scale of one million given a cluster of 1000 data nodes, 100 concurrent queries, and an average of 10 data exchange operators per query (1000 * 100 * 10 = 1,000,000). To overcome this challenge, we designed a unique communication service infrastructure where each data exchange communication pair between a consumer and a producer is considered a virtual or logical connection. Logical connections between a given pair of nodes share one physical connection. By virtualizing the data exchange connections with shared physical connections, the total number of physical connections on a physical host system is significantly reduced. We chose SCTP (Stream Control Transmission Protocol) as the transport layer protocol to leverage its built-in stream mechanism. SCTP offers reliable message-based transportation and allows up to 65535 streams to share one SCTP connection. In addition, SCTP can support out-of-band flow control with better behavior control and fairness. All those features match the requirements of our design of logical connections and simplify the implementation compared to the customized multiplexing mechanism.

2.1.2 High Availability and Replication

It is always a challenge to achieve high availability of database service across a large scale server fleet. Such system may encounter hardware failures so as to considerably impact service availability. FI-MPPDB utilizes primary-secondary model and synchronous replication. The amount of data stored in data warehouses are normally huge, up to hundreds of TB or even PB, so saving storage usage is a critical way to lower overall cost. A data copy is stored in primary and secondary data nodes, respectively. In addition, a dummy data node maintains a log-only copy to increase availability when secondary data nodes fail. Introducing the dummy data node solves two problems in synchronous replication and secondary data node catch-up. First, when secondary data nodes crash, primary data nodes can execute bulkload or DML operations because log can still be synchronously replicated to the dummy data nodes. Second, after recovering from crash, secondary data nodes need to catch up with primary data nodes for those updates happening when secondary data nodes are down. However, the primary data nodes may have already truncated log, causing secondary data nodes' recovery and catch-up to fail. This is solved by dummy data nodes providing the needed log.

2.1.3 Workload Management

The performance of analytical query processing is often sensitive to available system resources. FI-MPPDB depends on a workload manager to control the number of concurrently running queries. The workload manager optimizes system throughput while avoiding the slow-down caused by queries competing for system resources.

The workload manager consists of three main components: resource pools, workload groups, and a controller. Resource pools are used for allocating shared system resources, such as memory and disk I/O, to queries running in the system, and for setting various execution thresholds that determine how the queries are allowed to execute. All queries run in a resource pool, and the workload group is used for assigning the arriving queries to a resource pool. Workload groups are used to identify arriving queries through the source of the queries such as its application name. The controller evaluates queries and dynamically makes decisions on execution based on the query's resource demands (i.e., costs) and the system's available resources (i.e., capacity). A query starts executing if its estimated cost is not greater than the system's available capacity. Otherwise, the query is queued. Resource bookkeeping and feedback mechanisms are used in keeping tracking of the system's available capacity. The queued queries are de-queued and sent to the execution engine when the system's capacity becomes sufficient.

2.2 Online Expansion

Modern massively parallel processing database management systems (MPPDB) scale out by partitioning and distributing data to servers and running individual transactions in parallel. MPPDB can enlarge its storage and computation capacity by adding more servers. One of the important problems in such scale-out operation is how to distribute data to newly added servers. Typically, the distribution approach uses certain algorithms (such as hash functions, modulo, or round-robin) to compute a value from one column (called distribution column in a table). This value is used to determine which server (or database instance) stores the corresponding record. The result of those algorithms depends on the number of servers (or database instances) in the cluster. Adding new servers makes those algorithms invalid. A data re-distribution based on the number of servers, including newly added ones, is needed to restore the consistency between distribution algorithms' result and the actual location of records. In addition, hash distribution may be subjected to data skew where one or more servers are assigned significantly more data, causing them to run out of space or computing resource. In such cases, one can choose a different hashing function, re-compute the distribution map, and move data around to eliminate the skew and balance the load.

2.2.1 Solution Overview

A naive implementation of redistribution is to take the table offline, reorganize the data in place, and move relevant data to newly added nodes. During this process, the table cannot be accessed. Alternatively, one can create a shadow table and load it with the data while keeping the original table open for query. But until the data is redistributed to the new nodes, the distribution property does not hold among the new set of nodes. In order to make the table available for query during the redistribution process, one choice is to change table distribution property from hash to random as done in Greenplum [20]. Such an approach allows the data to be queried, but the query performance is degraded since data locality information is lost and collocated joins are not possible. In addition, data modification operations (such as IUD) are blocked on the table during redistribution, causing interruption to user workloads for extended period of time.

For our solution, we use the shadow table approach for redistribution. But instead of making the table read-only, we modify the storage property of the original table to append-only mode and prevent the recycling of the storage space. This gives us an easy way to identify the new records added to the table (called append-delta) during data redistribution. Additionally we create a temporary table to store the keys (rowid) of deleted records (called delete-delta). After the existing data has been redistributed, we lock down the

table, reapply the append-delta, and then delete-delta on the shadow table. To facilitate the application of delete-delta, the shadow table is created with additional column that stores the rowid from the original table. This way we can join the shadow table with delete-delta table to apply the delete-delta.

Our approach offers the following advantages

- Our solution allows DML operations including insert, delete, and update while the table is being redistributed.
- Our method of redistribution of a table can be configured to progress in small batches. Each batch can be done quickly to minimize load increase to the system. Each unit is done as a transaction and the whole redistribution can be suspended and resumed between tables.
- Our method requires only one scan of data in the original table for the redistribution. This improves the overall redistribution performance. The only additional scan of data is done on the shadow table when delete-delta is being applied.
- Our method can also be used for cluster downsizing without any changes. In both cases, extra space on each member of the new cluster is needed to store the temporary shadow table. The total size of the temporary shadow table is the same as the size of the original table.

2.2.2 *Core Algorithm*

Algorithm 1 illustrates the algorithm to execute the redistribution while still allowing normal DML operation on the table.

Algorithm 1 Algorithm_Redistribution_DML

1: Create a shadow table S with the same schema as the original table T to be redistributed
2: Mark T as append only
3: Disable garbage collection on T
4: Create a delete-delta table D for deletes on T
5: Redistribute a segment of T into S.
6: Apply D on S and reset D when finished.
7: Commit the change.
8: Repeat steps 5-7 until the remaining records in T is smaller than a threshold
9: Lock the original table. Redistribute the remaining insert and delete delta, just as in step 5 and 6.
10: Switch the T with S in the catalog.
11: Commit the changes.
12: Rebuild indexes

In Algorithm 1, steps 1 through 4 prepare the table T for redistribution. In step 1, we create a shadow table S with the same schema as T plus a hidden column (original_tuple_key) to store the original tuple key of records moved from T. Indexes on S are disabled during this phase until all redistribution is done. In step 2, we mark T as append-only and disable reuse of its freed space. With this conversion, inserts to T will be appended to the end of T, deletions are handled by marking the deleted record, and updates on the T are internally converted to deletions followed by inserts. In step 3, we disable the garbage collection on T to keep records

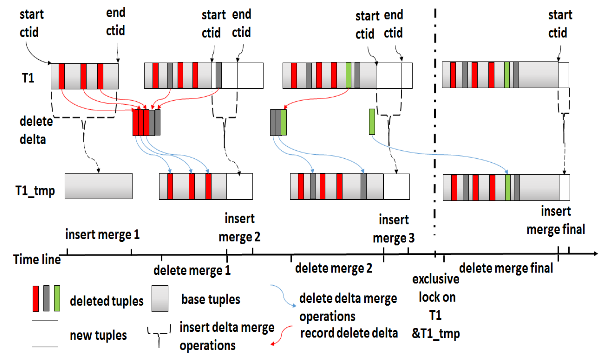

Figure 2: Example of Redistribution Phases

of its original position during the data redistribution process. In step 4, a temporary table delete-delta D is created to keep track of the original tuple keys of deleted records. Since the space in the original table is not reused after the redistribution has begun, the tuple key can uniquely identify a version of a record, allowing us to apply the deletions in the new table later on.

In steps 5 through 7 we start rounds of redistribution of records from T, one segment at a time. In step 5, we get a snapshot and start to scan a segment of the table, move all visible records to S, distribute them into a new set of nodes according to the new distribution property (new node group and/or new hash function), and record the unique original tuple key from the original table in the hidden column of the new table (explained in step one). In step 6, we apply the deletion in this segment happened during the redistribution of the segment by deleting the records in the new table using the original-tuple-key stored in D. D is reset after it is applied and this phase is committed in step 7.

In step 8, we iterate over steps 5 through 7 until the remaining data in T is small enough (based on some system threshold). Step 9 starts the last phase of the redistribution by locking T, redistributing the remaining data in T the same way as steps 5-6. This is followed by renaming the new table S to the original table T in step 10. Step 11 commits the changes. Finally, we rebuild the indexes on T in step 12.

Figure 2 illustrates the redistribution process from step 5-9 using an example with 3 iterations. In this example, T1 is the original table, and T1_tmp is the shadow table with the same schema as T1. All three iterations apply both insert and delete delta changes introduced by Insert, Update and Delete (IUD) operations of T1 onto T1_tmp while still taking IUDs requests against T1. The last iteration which has small set of delta changes locks on both tables exclusively in a brief moment to redistribute the remaining insert and delete delta as described in step 9.

2.3 Auto Tuning in Query Optimizer

Our MPPDB optimizer is based on Postgresql optimizer with fundamental enhancements to support complex OLAP workloads. We briefly describe the main architectural changes in our optimizer. First, our optimizer is re-engineered to be MPP aware that can build MPP plans and apply cost based optimizations that account for the cost of data exchange. Second, the optimizer is generalized to support

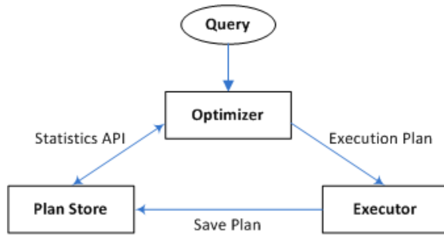

Figure 3: Statistics Learning Architecture

planning and cost based optimizations for vector executions and multiple file systems including Apache ORC file format. Query rewrite is another major ongoing enhancement to our optimizer, including establishing a query rewrite engine and adding additional rewrites which are critical to complex OLAP queries.

The enhancements mentioned above are common in other commercial database and big data platforms. We believe that we are closing the gap with those established products. In addition, we are working on cutting edge technology based on *learning* to make our optimizer more competitive. Our initial vision of a learning optimizer is in the area of cardinality estimation (statistics), which is one of the core components of cost based optimization. Traditional and classical methods of statistics used for cost based optimizations are complex, require large capital investment, and evolve through time. Different sources of data with their different formats pose additional challenges to collecting statistics with reasonable accuracy.

Based on the above challenges in the optimizer statistics area, we are developing a learning component in the DBMS that selectively captures actual execution statistics. The captured data in turn can be used by the optimizer for more accurate statistics estimates for subsequent similar queries. Our experience shows that most OLAP workloads are focused on specific queries/reports and therefore the approach of capturing execution plans and re-using them is promising and expected to mitigate the optimizer statistics limitations. This approach is also appealing, given that its engineering cost is much less than the traditional methods of collecting and using optimizer statistics, which it took decades to develop for the big players of OLAP and data warehousing, such as IBM DB2, Oracle and Teradata.

Figure 3 is a high level illustration of our statistics learning approach. The new architecture has two sub-components: capturing execution plans and re-using them by the optimizer. Based on user settings/directives, the execution engine (executor) selectively captures execution plan into a plan store. The plan store schema is modeled after the plan execution steps. Each step (scan, join, aggregation, etc) is captured with the estimated and actual row counts. To make the store more efficient and useful, the executor captures only those steps that have a big differential between actual and estimated row counts. A related but different store is general MPPDB plan store which is more detailed and used for auditing and off-line analysis.

The optimizer gets statistics information from the plan store and uses it instead of its own estimates. This is done efficiently through an API call to the plan store which is modeled as a cache. The key of the cache is encoded based on different steps description that includes step type, step predicate (if any), and input description. Obviously, the use of steps statistics is done opportunistically by the optimizer. If no relevant information can be found at the plan store, the optimizer proceeds with its own estimates. Our initial proof of concept for statistics learning is done for scan and join steps. We call this approach *selectivity matching*.

In addition to exact matches with previously stored predicates, auto tuning can be applied to similar predicates as well. We can gather predicate selectivity feedbacks in a special cache (separate from the plan store cache described above) and use it to estimate selectivity for similar conditions. Many machine or statistical learning algorithms can be used for this purpose. We call this second learning technique *similarity selectivity* and we also call this special case as *predicate cache*.

Our *similarity selectivity* model is initially applied to complex predicate like $x > y+c$ where both x and y are columns and c is a constant. Such complex predicates are common with date fields. For example, some of the TPC-H queries involve predicates like $l_receiptdate > l_commitdate + c$ which restricts line items that were received late by c days. These predicates pose a challenge to query optimizers and they are good candidates for our similarity selectivity. The more general form of these predicates is $x > y + c1$ and $x <= y + c2$. For example, $P1 = l_receiptdate > l_commitdate + 10$ and $l_receiptdate <= l_commitdate + 20$ retrieves all line items that are between 10 and 20 days late.

We choose KNN (K Nearest Neighbors) as the approach for our *similarity selectivity*. The selectivity of a new predicate is estimated to be the average selectivity of its K nearest neighbors in the predicate cache. The rationale here is that events close to each other have similar properties. For example, late line items tend to be fewer and fewer as the difference between the receipt date and commit date gets bigger. The distance metric used by KNN is simply the Euclidean distance based on $c1$ and $c2$. For example, the distance between $P1$ above and $P2 = l_receiptdate > l_commitdate$ and $l_receiptdate <= l_commitdate + 15$ is computed as the Euclidean distance between the two points (10,20) and (0,15).

2.4 MPPDB over HDFS (aka SQLonHDFS)

As big data platform and cloud become popular in recent years, many of our customers store huge amount of semi-structured and nonstructural data in Hadoop HDFS [11] in addition to storing their structured data in MPPDB. Hadoop HDFS is capable of scaling to petabytes of data and is steadily improving its performance and availability. One significant customer requirement is to be able to query data across heterogeneous data storage systems including HDFS. In this section, we discuss how our system addresses such needs.

2.4.1 SQLonHDFS using Foreign Data Wrapper

FI-MPPDB first adopted the HDFS as an external data source through PostgreSQL's Foreign Data Wrapper (FDW). We introduced a Scheduler component to bypass Hadoop Map-Reduce framework, and dynamically assigned splits of data files to MPP data nodes for efficient processing and load balancing. Similar architecture can be found on other open source SQLonHDFS solutions such as HAWQ [8] and Impala [24]. Figure 4 illustrates the flow of FI-MPPDB query processing:

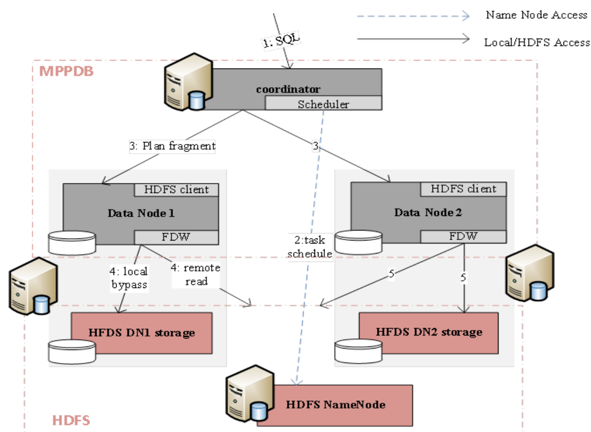

Figure 4: MPPDB Foreign HDFS Data Architecture

1. Our Gauss MPP coordinator receives the query in SQL format.

2. Planner constructs the query plan while Scheduler schedules tasks for each MPP data node according to table splits information from HDFS name node.

3. Coordinator ships the plan fragments with task-to-Datanode map to each MPP data node.

4. Each MPP data node reads data in a local-read preferred fashion from HDFS data nodes according to the plan and task map.

In general, a HDFS directory is mapped into a database foreign table which is stored like a native relational table in FI-MPPDB. Foreign tables can be defined as partitioned tables. A HDFS partition is normally stored as a HDFS directory that contains data sharing the same partition key. Both of our planner and scheduler can take advantage of this to generate plans that skip irrelevant partitions at run time to reduce I/O. Furthermore, some popular formats of HDFS file such as ORC or Parquet embed some level of indexes and synopsis within the file itself. Our planner and execution engine can leverage this information to push predicates down to file readers in order to further improve query performance. Other improvements we have done include leveraging our vectorized execution engine for efficient query execution and using dynamic multi-dimension runtime filter from star-join to further prune partitions and reduce data accessing from HDFS storage.

After its release, our SQLonHDFS feature becomes popular since our customers can directly query large amount of data inside HDFS without using ETL to reprocess and load data into FI-MPPDB. For further improvements, our customers make two important requirements:

1. Data collocation: as SQLonHDFS feature gets used for more critical analytics workloads, customers want better performance through data collocation.

2. DML/ACID support: as most of our customers migrate from commercial relational data warehouses where DML and ACID property are maintained, they expect similar functionalities from our system.

In the following sections, we will discuss our recent improvements in these areas.

2.4.2 *Advanced Data Collocation and Hash Partitioning*

HDFS doesn't support data collocation through consistent hashing algorithm which is a performance enhancement technique widely adopted in commercial MPPDB systems including FI-MPPDB. This means standard database operations such as JOIN, GROUP BY, etc will often require extra data shuffling among the clusters when performed on HDFS storage.

There are two kinds of data collocation schemes we considered:

1. Data collocation between MPPDB data nodes and HDFS data nodes: this allows MPP data nodes to scan data in HDFS through short-circuit local read interface where higher scan speed can be achieved.

2. Table collocations in HDFS data nodes: tables are partitioned on HDFS data nodes so that co-located join or group by operations can be performed to reduce network data shuffle cost.

When FI-MPPDB writes data to the HDFS system, it can apply consistent hash partition strategy. As illustrated in Figure 5, FI-MPPDB data nodes are co-located with the HDFS data nodes, where both local and distributed file system are presented. Direct local reads on HDFS can be done through the HDFS short-circuit read interface. When we load data into HDFS through our MPPDB data node, we use a local descriptor table to record each file's ownership within each data node. The descriptor table is shown as the Block Map in Figure 6. The table consists of columns for block id, min/max values for each columns in a block, a bitmap for deleted records in that block, and a block locater. Once the data is distributed to each MPP data node, it will serialize the incoming data stream into specific PAX style files such as ORC or Parquet and then write these files directly to the HDFS. The file replication is taken care of by HDFS itself. By default, three copies of the file are written. According to the file placement policy, one of the copies will be placed in the local machine, which is how the data co-location is achieved.

Note with this architecture, the block map dictates what HDFS files a node should access according to data partition strategy. It only preserves a logical HDFS locality of the files to their MPP worker nodes. The files in HDFS can be moved around by HDFS without notifying the MPP cluster due to storage re-balancing or node failures. To reduce the chance of remote file access, we use a HDFS hint to instruct HDFS name node to try its best to preserve the original location of specified files.

2.4.3 *DML support*

Since HDFS is append-only storage and not optimized for reading or manipulating small chucks of data, DML operations on HDFS can be complex and inefficient. We adopt a hybrid architecture to support DML operations by combining a write optimized row format of our MPPDB with a read optimized PAX format in HDFS. DML support addresses our customer's requirement of performing occasional DML operations on data in HDFS cluster with strong ACID

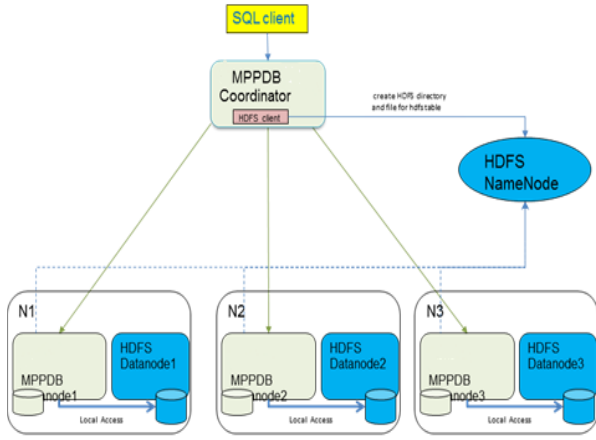

Figure 5: MPPDB Native HDFS Table Architecture

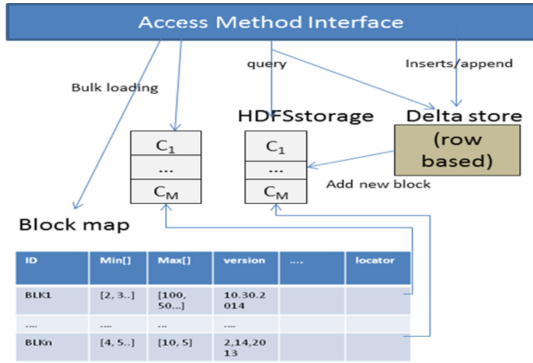

Figure 6: Native HDFS Table Representation

property. The visibility of block map determines the visibility of the rows in a block. Specifically, we store inserted rows first in a row-based delta table. Only after the delta table reaches certain size, it will then be converted and written into the HDFS as read optimized PAX format. For the delete operation, there are two scenarios: 1) if a deleted row is found in the local delta table, it can be deleted from there immediately; 2) if the deleted row is found in a PAX file, we mark the row deleted in the corresponding bitmap of the block map. In this way, we avoid physically rewriting the whole file on HDFS before a delete commits. Compaction can be done later to physically re-construct the data based on the visibility information recorded in the block map when there are enough rows deleted. An update operation is simply a delete of the existing rows followed by an insert of new rows into the delta table. The access interfaces are illustrated in Figure 6.

2.5 Intelligent JIT Compiled Execution

In our FI-MPPDB, we designed and implemented the Just-In-Time (JIT) compiled query execution using LLVM compiler (for short we just refer to it as JIT) infrastructure. The JIT compiled code targets CPU intensive operators in our query execution engine. Our goal is to produce JIT compiled code with less instructions, less branches, less function calls, and less memory access compared with the previous approach of interpreted operator execution. The JIT compilation can be extended further to cover more compiler optimizations, such as loop unrolling, function in-lining, constant propagation, and vectorization *etc.* In addition, some frequently used database functions can be replaced by hardware instructions. For example, CRC32 instruction in x86 hardware can be used as a hash function in various operators (hash join, hash aggregation), and the hardware overflow flag can replace the software code for integer overflow check.

JIT compiled execution may not always be better than the standard code interpretation approach. This could be the case for small data sets where the overhead of JIT compilation is more than the execution speedup. Also, additional optimizations can be applied on the JIT compiled code. Examples of such optimizations include function in-lining, loop unrolling, constant propagation, and vectorization. Such optimizations also have a trade-off between the optimization overhead and the speedup they provide. We tackle these trade-offs by making an intelligent decision among these three possibilities: (1) use interpreted code without JIT code generation, (2) JIT code generation without additional optimizations (3) JIT code generation with additional code optimizations.

For a specific query, the cost model (for intelligent decision making) chooses no code generation if the data size of the query is small and the performance gain from the generated code is less than the JIT compilation cost. If the workload size is large enough, the cost model will choose the optimal method to generate the most cost effective code based on the data size and the cost of JIT compilation. Our cost model is based on the formula (1) below to estimate the performance gains by applying a specific method of JIT compiled execution on a specific function or a piece of code.

$$P = (T_1 - T_2) \times N - T_{JITcost} \qquad (1)$$

In the formula (1) shown above, T_1 is the cost of one time execution on the original code, T_2 the cost of one time execution on the JIT compiled code, N the data size of the workload, $T_{JITcost}$ the JIT compilation cost, and P the performance gain. The estimation of execution cost is similar to the cost model applied in the database optimizer. The JIT compilation cost is estimated according to the size of the generated code. Our experiment results show that the actual JIT compilation cost is proportional to the size of the generated code.

For a specific query, according to the cost model, the performance gain is linearly proportional to the data size of workloads. Suppose, we have two methods of JIT compiled execution for this query, the formula (2) below illustrates how to choose different methods of JIT compiled execution according to different size of workloads. If the workload size is not larger than N_1, we do not apply the JIT compiled execution on this query. If the workload size is between N_1 and N_2, we apply the method with less optimizations and less JIT compilation cost. If the workload size is larger than N_2, we apply the method with more optimizations and more JIT compilation cost to achieve better performance.

$$P = \begin{cases} 0 & (x \leq N_1), \\ a_1 x + b_1 & (N_1 < x \leq N_2), \\ a_2 x + b_2 & (x > N_2). \end{cases} \qquad (2)$$

In the following, we use two queries extracted from real customer use cases to illustrate how our cost model chooses the optimal method of JIT compiled execution. Without loss of generality, these methods of code generation and the cost model can be applied to many other operators of execution engine in a database system, for example, IN expression, CASE WHEN expression, and other aggregation functions etc.

4-SUM query: SELECT SUM(C1), ... , SUM(C4) FROM table;

48-SUM query: SELECT SUM(C1), ... , SUM(C48) FROM table;

For each of the above two queries, we have two methods of code generation to apply JIT compiled execution. In the first method, we applied various optimizations including loop unrolling, function in-lining, and more code specialization, to generate the code. In the second method, we generate the code without optimization. The first method generates more efficient code but consumes more time to produce. The second method has less optimized code but requires less time to make compared to the first method. Our experiment results in section 3.5 show that our cost model picks the optimal choice between these two options for the two queries.

3. EXPERIMENTAL RESULTS

In this section, we present and discuss experimental results of our work in the four areas described in section 2.

3.1 Online Expansion

Our online expansion tests start with a cluster of 3 physical machines, each of which has 516GB system memory, Intel Xeon CPU E7-4890 v2 @ 2.80 Ghz with 120 cores, and SSD driver. The actual expansion tests add 3 more physical machines with the same configuration. We thought OLAP was more appropriate for our test and we decided to use the TPC-DS in our test. We loaded the cluster with TPC-DS data with scale factor 1000 (1 Terabyte) and we ran all the 99 TPC-DS queries. We also used the TPC-DS data maintenance tests for inserts, updates, and deletes (IUDs). Each transaction of IUD modifies or inserts an average of 500-1000 rows. We used a total of 5 threads running on the cluster: one for online redistribution, one for queries, and three for IUDs.

Figure 7 captures the results of our online expansion tests. We have conducted two major tests. The first test is covered by the first 3 bars in Figure 7 (bars *a*, *b* and *c*). Bar *a* is the total elapsed time for customer application workload (including query and IUD operations) during online expansion using our new online expansion method. Bar *b* and *c* are the total elapsed time for the same workload but using offline expansion method at different points of time. In *b*, the workload is executed at the old cluster first followed by the offline cluster expansion, while *c* is the opposite where the expansion is followed by workload execution. The total elapsed time using our new online expansion (bar *a*) is a lot better than the workload first then offline expansion (bar *b*) and close to the method of offline expansion first and then running workload (bar *c*). Note that the result in bar a is actually a lot better than those in bar c because it has a better response time since user requests can be processed during the expansion. The cost of online expansion is more than offline expansion which is expected.

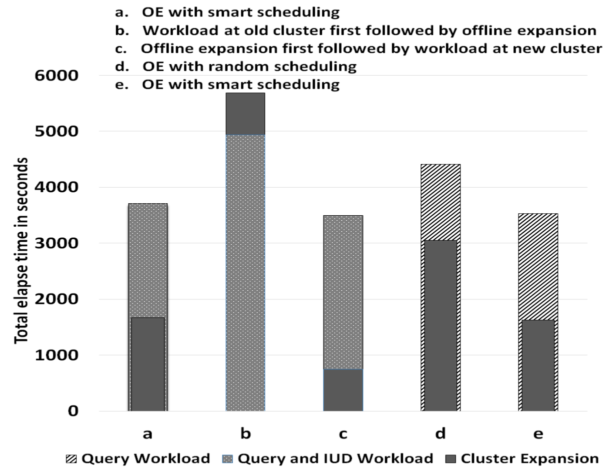

Figure 7: TPC-DS 1000x Query and IUD Workload with Online Expansion

Next, we compare online expansion performance to show the effect of scheduling the expansion order of tables in bar *e* and *d*. For simplicity, we only executed query workload (no IUD) during online expansion time. Bar *d* is the elapsed time for query workload during online expansion with random scheduling which does the table redistribution in random order. Smart scheduling (bar *e*) on the other hand orders the tables for redistribution according to the order of tables in the query workload. The smart order basically tried to minimize the interaction between the redistribution and workload to speed up both processes. Comparing the result from *e* with that in bar *d*, a schedule considering the access pattern improves data redistribution performance by roughly 2X, and improves the query performance by about 20%.

3.2 Auto Tuning in Query Optimizer

We prototyped capturing execution plans and re-using them automatically by our optimizer. We conducted our testing on a 8-nodes cluster running FI-MPPDB. Each node is a Huawei 2288 HDP server with dual Intel Xeon eight-core processors at 2.7GHz, 96GB of RAM and 2.7 TB of disk running with CentOS 6.6. We loaded the cluster with 1 Terabyte of TPC-H data with two variants of the *lineitem* and *orders* tables. The first variant has both tables hash partitioned on *orderkey* to facilitate co-located joins between them. The other variant has the *lineitem* table hash partitioned on part key and the *orders* table hash partitioned on customer key. This variant is used to test the impact of selectivity on network cost.

The next three subsections cover three tests that aim at testing the impact of inaccurate predicate selectivity for table scans. The three experiments are: wrong choice of hash table source in hash joins, unnecessary data movement for parallel execution plans, and insufficient number of hash buckets for hash aggregation. The experiments are based on a query shown below.

```
select o_orderpriority, count(*) as ct
from lineitem, orders
where l_orderkey=o_orderkey and
l_receiptdate <op> l_commitdate + date ':?';
```

Table 1: Comparison of actual and estimated selectivity for late line items predicate

Predicate Description	Predicate	Actual Selectivity	Estimated Selectivity
line items received more than 60 days late	$l_receiptdata > l_commitdata + 60$	15%	33%
line items received more than 120 days late	$l_receiptdata > l_commitdata + 120$	0.0005%	33%
line items received 40 days early or less	$l_receiptdata \leq l_commitdata - 40$	8%	33%
line items received 80 days early or less	$l_receiptdata \leq l_commitdata - 80$	0.05%	33%

Note, <op> is '<=' for early line items **and** '>' for late line items

The above query checks late/early line items based on difference between receipt and committed dates. The predicate that checks how early/late a line item like $l_receiptdata > l_commitdata + 120$ poses a challenge to query optimizers and we thought it is a good example of statistics learning. Most query optimizers use a default selectivity for such predicates (MPPDB use 1/3 as the default selectivity). Table 1 shows actual and estimated selectivity of these predicates for different values of being late or early.

3.2.1 Hash join plan problem

We ran the early/late line items query using the different options for being early/late per the entries in Table 1. Figure 8 illustrates the performance difference between the current optimizer and our prototype for the early/late order query. The join in this query does not require shuffling data since both tables are co-located on the join key (this is the first variant). Assuming hash join is the best join method in this case, choosing the right hash table build is the most critical aspect of the plan for this query. The *orders* table is 1/4 of the *lineitem* table, and the current optimizer assumes 1/3 of the rows in *lineitem* satisfying the early/late predicate. This leads the current optimizer to choose the *orders* table as the hash build side for all the variations of the query.

The inaccurate estimate of the selectivity did not have an effect on the plan for line items with relatively close values of *l_receiptdate* and *l_commitdate* (line 1,2 and 5 in Table 1). The reason is that the *orders* table is still smaller than the *lineitem* table for those cases and placing it in the hash table is a good execution plan. The other cases of too late/early (lines 3, 4, 6 and 7 in Table 1) have the opposite effect with smaller *lineitem* table, and in those cases the auto tuning estimate outperformed the standard estimate.

3.2.2 Unnecessary data shuffle

Inaccurate predicate selectivity could also have an impact on the amount of data shuffled in MPP join plans which are typically used to place matching join keys on the same node. This problem is tested by running the same early/late line items query discussed in the previous section on the second variant of *lineitem* and *orders* tables where they are not co-located on the join key. With this data partitioning, the optimizer has three options to co-locate the join keys and perform the join in parallel on the data nodes. These options are: re-partition both tables on *orderkey*, replicate *orders* table, or replicate *lineitem* table. Note that big data engines always have these three options for joining tables since HDFS data are distributed using round robin and thus join children are not co-located on the join key.

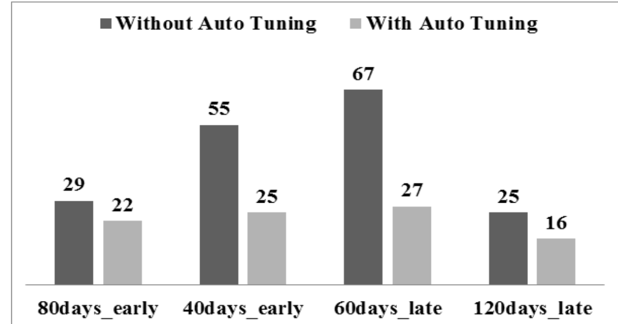

Figure 8: Performance comparison on early/late received orders query (in min.)

The query optimizer chose the shuffle option (among the three above) which involves the least amount of data movement to reduce the network cost. The standard optimizer elects to re-partition both *lineitem* and *orders* tables (option 1 above) for all variations of the early/late line item query. This is the case since the size estimates of both tables are close. This plan is coincidentally the best plan for three cases of the variations (line 1,2 and 5 in Table 1) of the early/late line items query. However, this plan is not optimal for the other cases where the filtered *lineitem* is smaller than *orders* table. Figure 9 captures the run-time for this case of early/late line items query. The performance difference is more compelling than those in Figure 8 since it includes both the extra data shuffle and the original hash table plan problem.

3.2.3 Insufficient hash buckets for hash aggregation

For the third and last test, we ran the official TPC-H Q4. As explained in the introduction section, The performance of the plan of TPC-H is sensitive to the size of the hash aggregation table set by the optimizer. The hash table size is set based on the number of distinct values for *l_orderkey* which is impacted by selectivity of $l_receiptdate > l_commitdate$. The standard default selectivity of "1/3" produced underestimates on number of rows and thus a lower estimate on the number of distinct values which misled the optimizer to use less hash buckets than needed for the hash aggregate step. This resulted in almost 2X slowdown (53 seconds vs 28

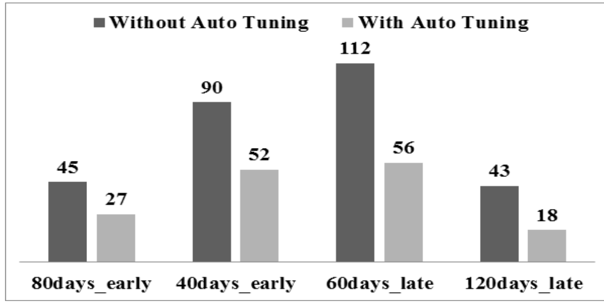

Figure 9: Performance comparison on early/late received orders query with shuffle (in min.)

Table 2: Selectivity Estimation Error for Early/Late Line Items

Cache Size	Cache Hits	KNN Error
100	218	1.2%
50	118	2.0%
25	65	3.6%

seconds) compared to the optimal number of hash buckets computed using our prototype.

3.2.4 Join Selectivity Experiment

The previous sub-sections cover in details three scenarios where selectivity learning for table scans has significant impact on query performance. We also did one experiment for learning join predicates selectivity. The experiment is based on the query shown below.

```
select count(*)
from lineitem, orders, customer
where l_orderkey=o_orderkey and
l_shipdate >= o_orderdate +
interval '121 days'
and o_custkey=c_custkey
and c_acctbal between
(select 0.9*min(c_acctbal) from customer) and
(select 0.9*max(c_acctbal) from customer)
group by c_mktsegment;
```

The query above finds those line items that took 121 days or more to process and handle. The query excludes top and bottom 10% customers in terms of their account balance. The query runs in 425 seconds with our current optimizer. The execution plan is not optimal and it joins customer and orders first and then joins the result to line item table. The root cause is that the optimizer uses 33% as the default selectivity for $l_shipdate \geq o_orderdate + interval'121day'$ instead of the actual 1% selectivity. Using our learning component, we get the optimal plan which runs in 44 seconds with more than 9X improvement.

3.3 Cache Model-based Selectivity Estimation

In this section, we demonstrate the use of KNN for our similarity selectivity approach. We tried varying K and found that K = 5 worked well for various cases. In this approach, the selectivity of a predicate is estimated to be the average selectivity from its 5 nearest neighbors in the predicate cache. We pick two predicates for experiment: one for early/late items and the other for normal items. The

Table 3: Specific Selectivity Estimation Error for Early/Late Line Items

Cache Size	c = -80 selectivity = 0.05%	c = -40 8%	c = 60 15%	c = 120 0.0005%
100	0.2%	0%	0%	0.05%
50	0.7%	-0.2%	3.8%	0%
25	5.0%	0.8%	-4.6%	0.8%

Table 4: Selectivity Estimation Error for Normal Line Items

Cache Size	Cache Hits	KNN Error
100	4	3.1%
50	1	4.0%
25	1	6.0%

former predicate is modeled with a single parameter (how early/late) in the predicate cache while the latter by two. Overall, our experiments show that the KNN approach is highly accurate in selectivity estimation: the absolute estimate errors are no more than 4% for single parameter case and no more than 6% for two parameters case. The details are described below.

We use the lineitem table from the TPC-H which has 6M rows. We focus on the early items satisfying the condition $l_receiptdate \leq l_commitdate - c$ days, and late items satisfying the condition $l_receiptdate > l_commitdate + c$ days. We can combine early and late items and model them with a single predicate cache: the parameter is the difference between *l_receiptdate* and *l_commitdate* with early items having negative differences (c) and late items positive ones. We randomly generate 500 predicates with c between -80 and 120. The average selectivity of such predicates is found to be 18%. For each predicate, we find its actual selectivity, use the KNN to estimate its selectivity, and compute the estimation error. We repeat this experiment with different cache sizes of 100, 50, and 25. The average absolute estimation errors for the 500 tests are shown in Table 2. We can see that larger caches provide better results but even with the smallest cache, the error is still pretty small (3.6%). Note that the cache is seeded with a random c before it is used for selectivity estimation. When the cache is full the LRU policy is used. The predicates only have 200 possible values for c so the cache hits are relatively high. For example, with a size of 25, the cache stores 1/8 of the possible values and there are 65 cache hits among the 500 tests. To see the KNN in use, we test it with specific predicates from Table 1 where c is in $\{-80, -40, 60, 120\}$. The estimated selectivity errors (negatives are under-estimation) are shown in Table 3. Overall, the estimation is pretty accurate with the biggest errors at about 5% with a cache size of 25.

To demonstrate the effectiveness of KNN with 2 parameters, we also do experiments for the normal line items: predicate $l_receiptdate$ *between* $l_commitdate - c1$ days *and* $l_commitdate + c2$ days are generated with two uniformly distributed variables $1 \leq c1 \leq 80$ and $1 \leq c2 \leq 120$. The average selectivity of such predicates is 67%. The results are shown in Table 4. The estimation errors are pretty small relative to the average selectivity of 67%. Note that the cache

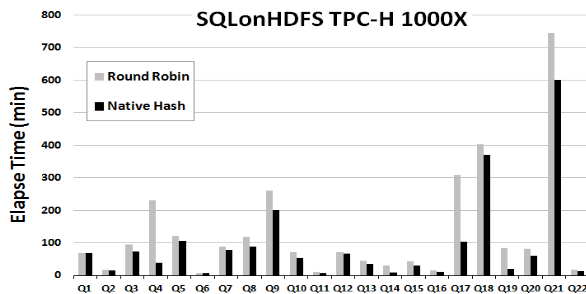

Figure 10: Performance comparison

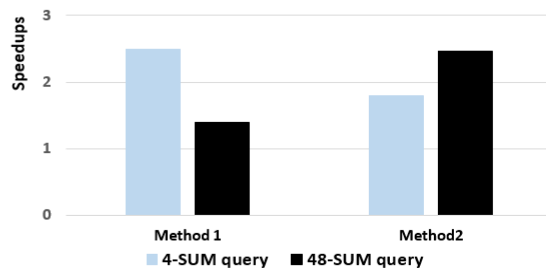

Figure 11: Speedups on different queries by applying different methods of code generation

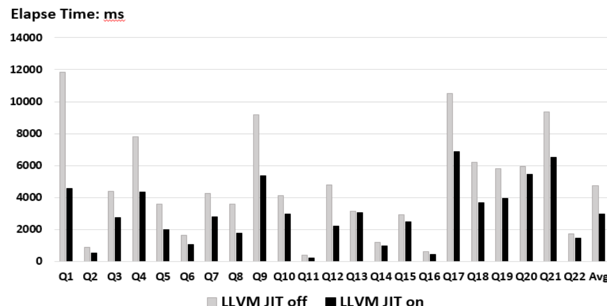

Figure 12: Performance gains from JIT compiled execution on TPC-H

hits are much less as there are 9600 possible combinations of $c1$ and $c2$. With the smallest cache size of 25, which is only 0.3% of possible combinations with a single cache hit, the estimation error is only 6%.

As stated above, we start using the cache for selectivity estimation when it only has a single predicate. This is a valid approach as the single selectivity is likely to gives us a estimate for similar predicates. This is probably better than a default selectivity such as 33%. Our experiments show that the cache estimation errors stabilize quickly as more feedbacks are available. In practice, we can allow the user to control the use of the cache for unseen conditions, e.g., use the cache for prediction only when it has entries more than a threshold. The cache can also keep the estimation errors, and the use of it can be shut down if the errors are excessively large for a certain number of estimations. This can happen when there are very frequent updates on the predicate columns.

3.4 Advanced Data Collocation Performance Improvement for SQLonHDFS

As described in section 2.4, we enhanced our SQLonHDFS solution by applying the FI-MPPDB hash distribution strategy to the HDFS tables and leverage upon the data distribution property through SQL optimizer to produce efficient plans. We conducted performance testing on an 8-node cluster running FI-MPPDB with the HDFS storage to demonstrate the performance improvement. Each node is a Huawei 2288 HDP server with dual Intel Xeon eight-core processors at 2.7GHz, 96GB of RAM and 2.7 TB of disk running with CentOS 6.6. We loaded the cluster with 1 terabyte of TPC-H data in two flavors: one data set is randomly distributed while the other data set is hash distributed based on distribution columns. In the second case, the distribution columns are carefully selected for each TPC-H table according to the popular join columns in order to avoid expensive data shuffling for the workload.

Figure 10 shows the results running the 22 TPC-H queries on the two data sets with different distribution properties. By taking advantage of data collocation, hash distribution clearly provides much better performance by lowering the network shuffling cost. It obtains over 30% improvement (geometric mean) in query elapsed time.

3.5 Intelligent JIT Compiled Execution

Our experiment setup is a single node cluster with one coordinator and two data nodes, and the server node is a Huawei 2288 HDP server with dual Intel Xeon eight-core processors at 2.7GHz, 96GB of RAM and 2.7 TB of disk running with CentOS 6.6. We tested our intelligent JIT compilation strategy on the two queries mentioned in section 2.5: 4-SUM and 48-SUM queries. The queries are extracted from actual Huawei use cases, and the table has about 100 million rows. Figure 11 shows the results for these queries. The optimal solution for 4-SUM is the first method of code generation with LLVM post-IR optimization while the speed up is better for 48-SUM query without LLVM post-IR optimizations. For both queries, our intelligent decision making system (cost model) selected the optimal solution.

Our more comprehensive test is based on the TPC-H workload with 10X data. Figure 12 shows that our JIT compiled execution approach outperforms the previous approach with no code generation. The average performance improvement on the 22 queries is 29% (geometric mean).

4. RELATED WORK

In this section, we discuss some work related to the four major technical areas presented in this paper.

Online Expansion We could not find in literature about online cluster expansion over large data warehouses with concurrent DML operations without downtime. The closest known work is about online expansion in Greenplum [20] which supports redistributing data efficiently without noticeable downtime while guaranteeing transaction consistency. However, their solution blocks DML operations during online expansion because their data redistribution command uses exclusive table locks during expansion. Amazon Redshift [2] also provides cluster resizing operation. However, it puts the old cluster in read-only mode while a new cluster is being provisioned and populated. Snowflake [15] takes a different approach in cluster expansion that allows

users to dynamically resize the warehouse with predefined size such as SMALL , MEDIUM, and LARGE etc. Carlos et al. [21] provide a design tool, FINDER, that optimizes data placement decisions for a database schema with respect to a given query workload.

Auto Tuning Early known work on database auto tuning include Database Tuning Advisor [18] from Microsoft SQL Server and LEO [30] from IBM DB2 system. LEO is a learning based optimizer that focuses on predicate selectivities and column correlation but does not cover the general case of operator selectivities. Our learning approach is simpler and more general since it captures actual and estimates for all steps and re-use the information for exact or similar steps in future query planning.

Using machine learning in query optimization, performance modeling, and system tuning has increasingly drawn attention to the database community [32, 19]. Recent works [22, 25, 26] use machine learning methods to estimate cardinality for various database operators. The approach using machine learning technique is promising as it allows the system to learn the data distribution and then estimate the selectivity or cardinality. In contrast, the traditional statistics based selectivity estimation has its limitation on the information a system can store and use. However, we need to carefully assess the benefits of adopting machine learning based approaches in commercial database query optimizer as they may require significant system architecture changes. For example, [25, 26] use neural networks to model selectivity functions which require significant training to approximate the functions. In addition, the training set takes considerable system resources to generate. In comparison, our similarity selectivity approach uses query feedbacks where no training is necessary. Our approach is close to [22] where KNN is used. We keep it simple by gathering similar feedbacks into a single predicate cache. The one to one mapping between predicate types and predicate caches makes it easy to integrate the approach into optimizer architecture.

SQL on HDFS Apache HAWQ [8] was originally developed out of Pivotal Greenplum database [10], a database management system for big data analytics. HAWQ was open sourced a few years ago and currently incubated within Apache community. With its MPP architecture and robust SQL engine from Greenplum, HAWQ implements advanced access library to HDFS and YARN to provide high performance to HDFS and external storage system, such as HBase [4]. SQL Server PDW allows users to manage and query data stored inside a Hadoop cluster using the SQL query language [29]. In addition, it supports indices for data stored in HDFS for efficient query processing [28].

With the success of AWS Redshift [2], Amazon recently announced Spectrum [3]. One of the most attractive features of Spectrum is its ability to directly query over many open data formats including ORC [6], Parquet [7] and CSV. FI-MPPDB also supports ORC and CSV formats, while neither Redshift nor Spectrum supports HDFS.

In the Hadoop ecosystem, Hive [5] and Impala [24] are popular systems. SparkSQL [16] starts with the focus on Hadoop, then expands its scope onto S3 of AWS. While they are popular in open source and Hadoop community, the lack of SQL compatibility and an advanced query optimizer make it harder to enter enterprise BI markets. In contrast, our SQLonHDFS system can run all 99 TPC-DS queries without modification and achieve good performance.

Intelligent JIT Compiled Execution CPU cost of query execution is becoming more critical in modern database systems. To improve CPU performance, more and more database systems (especially data warehouse and big data systems) adopt JIT compiled query execution. Amazon Redshift [2] and MemSQL (prior to V5) [12] transform an incoming query into C/C++ program, and then compile the generated C/C++ program to executable code. The compiled code is cached to save compilation cost for future execution of the query since the cost of compilation on a C/C++ program is usually high [27]. In comparison, the LLVM JIT compilation cost is much lower so there is no need to cache the compiled code. Cloudera Impala [24] and Hyper system [27] apply JIT compiled execution using LLVM compiler infrastructure, but they only have a primitive cost model to decide whether applying the JIT compilation or not. Spark 2.0's [16] Tungsten engine emits optimized bytecode at run time that collapses an incoming query into a single function. VectorWise database system [23] applies the JIT compiled execution as well as the vectorized execution for analytical database workloads on modern CPUs. Our FI-MPPDB adopts JIT compiled execution using LLVM compiler infrastructure to intelligently generate the optimal code based on our cost model.

5. FUTURE WORK AND CONCLUSION

In this paper, we have presented four recent technical advancements to provide online expansion, auto tuning, heterogeneous query capability and intelligent JIT compiled executions in FI-MPPDB. It is a challenging task to build a highly available, performant and autonomous enterprise data analytics platform for both on-premise and on the cloud.

Due to space limit, we briefly mention two of our future works. First, cloud storage service such as AWS S3, Microsoft Azure Storage, as well as Huawei cloud's OBS (Object Block Storage) have been widely used. Users often want to directly query data from the centralized cloud storage without ETLing them into their data warehouse. Based on customer feedback, we are currently working on providing SQLonOBS feature in our cloud data warehouse service based on the similar idea of SQLonHadoop described in this paper. Second, auto tuning database performance using machine learning based techniques is just starting but gaining heavy momentum. In addition to selectivity learning, we plan to look into parameter learning, where key parameters the query optimizer uses to make decisions can be automatically learned. We also plan to explore other machine learning techniques.

6. ACKNOWLEDGMENT

We thank Qingqing Zhou, Gene Zhang and Ben Yang for building solid foundation for FI-MPPDB, and Harry Li, Lei Liu, Mason Sharp and other former team members for their contribution to this project while they worked at Huawei America Research Center. We also thank Huawei headquarter team for turning these technologies into a successful product on the market.

7. REFERENCES

[1] Amazon Athena - Amazon. https://aws.amazon.com/athena/.

[2] Amazon Redshift - Amazon. https://aws.amazon.com/redshift/.

[3] Amazon Redshift Spectrum - Amazon. https://aws.amazon.com/redshift/spectrum/.

[4] Apache - HBase. https://hbase.apache.org/.

[5] Apache - Hive. https://hive.apache.org/.

[6] Apache - ORC. https://orc.apache.org/ .

[7] Apache - Parquet. https://parquet.apache.org/.

[8] Apache HAWQ - Apache. http://spark.apache.org/sql/.

[9] FusionInsight MPPDB - Huawei. http://e.huawei.com/us/products/cloud-computing-dc/cloud-computing/bigdata/fusioninsight.

[10] Greenplum Database - Pivotal. http://www.greenplum.com.

[11] Hadoop - Apache. http://hadoop.apache.org/ .

[12] MemSQL. https://www.memsql.com/.

[13] Oracle Autonomous Database Strategy White Paper. http://www.oracle.com/us/products/database/autonomous-database-strategy-wp-4124741.pdf.

[14] Postgres-XC. https://sourceforge.net/projects/postgres-xc/.

[15] Snowflake - Snowflake. https://www.snowflake.net/.

[16] Spark SQL & DataFrames - Apache. http://spark.apache.org/sql/.

[17] Transwarp Inceptor. http://www.transwarp.cn/.

[18] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the 30th VLDB Conference, Toronto, Canada*, pages 1110–1121, 2004.

[19] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of SIGMOD 2017, Chicago, USA*, pages 1009–1024, 2017.

[20] J. Cohen, J. Eshleman, B. Hagenbuch, J. Ken, C. Pedrotti, G. Sherry, and F. Waas. Online Expansion of Large-scale Data Warehouses. In *PVLDB, 4(12):1249-1259*, 2011.

[21] C. Garcia-Alvarado, V. Raghavan, S. Narayanan, and F. M. Waas. Automatic Data Placement in MPP Databases. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, 2012.

[22] O. Ivanov and S. Bartunov. Adaptive Query Optimization in PostgreSQL. In *PGCon 2017 Conference, Ottawa, Canada*, 2017.

[23] M. Z. J. Sompolski and P. Boncz. Vectorization vs. compilation in query execution. In *In Proc. of the 7th International Workshop on Data Management on New Hardware*, 2011.

[24] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-milne, and M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *In Proc. CIDR15*, 2015.

[25] H. Liu, M. Xu, Z. Yu, V. Corvinell, and C. Zuzarte. Cardinality Estimation using Neural Networks. In *Proceeds of the 25th Annual International Conference on Computer Science and Software Engineering (CASCON15)*, pages 53–59, 2015.

[26] H. Lu and R. Setiono. Effective Query Size Estimation Using Neural Networks. In *Applied Intelligenve*, pages 173–183, May 2002.

[27] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In *PVLDB, 4(9):539-550*, 2011.

[28] V. Reddy Gankidi, N. Teletia, J. Patel, A. Halverson, and D. J. DeWitt. Indexing hdfs data in pdw: Splitting the data from the index. In *PVLDB, 7(13):1520-1528*, 2014.

[29] S. Shankar, R. Nehme, J. Aguilar-Saborit, A. Chung, M. Elhemali, A. Halverson, E. Robinson, M. Subramanian, D. DeWitt, and C. Galindo-Legaria. Query Optimization in Microsoft SQL Server PDW. In *Proceedings of SIGMOD 2012, Scottsdale, USA*, pages 767–776, 2012.

[30] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2s LEarning Optimizer. In *Proceedings of the 27th VLDB Conference, Roma, Italy*, 2001.

[31] F. Waas. Beyond Conventional Data Warehousing -Massively Parallel Data Processing with Greenplum Database. In *In Proc. BIRTE*, 2008.

[32] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi, and K.-L. Tan. Database Meets Deep Learning: Challenges and Opportunities. In *SIGMOD Record, Vol. 45, No2*, 2016.