HAWQ: A Massively Parallel Processing SQL Engine in Hadoop

Lei Chang, Zhanwei Wang, Tao Ma, Lirong Jian, Lili Ma, Alon Goldshuv
Luke Lonergan, Jeffrey Cohen, Caleb Welton, Gavin Sherry, Milind Bhandarkar
Pivotal Inc
{lchang, zwang, tma, ljian, lma, agoldshuv,
llonergan, jcohen, cwelton, gsherry, mbhandarkar}@gopivotal.com

ABSTRACT

HAWQ, developed at Pivotal, is a massively parallel processing SQL engine sitting on top of HDFS. As a hybrid of MPP database and Hadoop, it inherits the merits from both parties. It adopts a layered architecture and relies on the distributed file system for data replication and fault tolerance. In addition, it is standard SQL compliant, and unlike other SQL engines on Hadoop, it is fully transactional. This paper presents the novel design of HAWQ, including query processing, the scalable software interconnect based on UDP protocol, transaction management, fault tolerance, read optimized storage, the extensible framework for supporting various popular Hadoop based data stores and formats, and various optimization choices we considered to enhance the query performance. The extensive performance study shows that HAWQ is about 40x faster than Stinger, which is reported 35x-45x faster than the original Hive.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: System—*parallel databases*

General Terms

Database, Hadoop, Query Processing

1. INTRODUCTION

The decreasing cost of storing data in Hadoop [1] is continuously changing how enterprises address data storage and processing needs. More and more Pivotal Hadoop customers are using HDFS as a central repository (aka, data lake) for all their history data from diverse data sources, for example, operational systems, web, sensors and smart devices. And unlike traditional enterprise data warehouses, data can be both structured and unstructured. To take maximum advantage of big data, the customers are favoring a new way of capturing, organizing and analyzing data. In the new way, instead of carefully filtering and transforming the source data through a heavy ETL process to data warehouses designed specifically for predefined analysis applications, the original data from data sources with few or no transformation is directly ingested into the central repository. The new paradigm increases the agility of new analytical application development. For example, when in need of some new data in new applications, analytic users do not need to go back to the data sources, change the ETL process or develop new ETL processes to get what they need but missed or filtered in the rigid design that is optimized for existing known analytical applications. And obviously, the new paradigm also gives data scientists more flexibility at ad-hoc analysis and exploration of the data.

The following are the typical requirements from customers who are willing to use Hadoop for their daily analytical work.

- **Interactive queries**: Interactive query response time is the key to promote data exploration, rapid prototyping and other tasks.
- **Scalability**: Linear scalability is necessary for the explosive growth of data size.
- **Consistency**: Transaction makes sense and it takes the consistency responsibility away from application developers.
- **Extensibility**: The system should support common popular data formats, such as plain text, SequenceFile, and new formats.
- **Standard compliance**: Keeping the investment in existing BI and visualization tools requires standard compliance, for example, SQL and various other BI standards.
- **Productivity**: It is favorable to make use of existing skill sets in the organization without losing productivity.

Currently, Hadoop software stack is still hardly satisfying all of the above requirements. MapReduce [19, 20] is the basic working horse behind the processing of both structured and unstructured data. It runs on commodity hardware, and it is scalable and fault tolerant. However, it has many known limitations, for example, not being suitable for interactive analytics due to its poor performance [27, 29, 32], and low-level programming model makes it not friendly for business analysts. Although some systems such as Hive [36, 37] and Pig [30] are developed to alleviate these issues, the performance is still not satisfactory for real-time data exploration. In addition, data consistency is a nightmare from application developer side. Without the transaction capability provided by infrastructure software, application programming is quite error prone and often leads to ugly and unmaintainable code [18, 34].

In database community, one of the major recent trends is the wide adoption of Massively Parallel Processing (MPP) systems [4, 6, 11], which are distributed systems consisting of a cluster of independent nodes. MPP database shares many good features with Hadoop, such as scalable shared-nothing architecture. And it is particularly excellent at structured data processing, fast query processing capability (orders of magnitude faster than MapReduce and

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2595636

Hive [27, 32]), embedded automatic query optimization and industry SQL standard compatible interfaces with BI tools, and it is particularly easy to use for data analysts.

Motivated by the limitations of current Hadoop and performance advantages of MPP databases, we developed HAWQ, which is a SQL query engine that combines the merits of Greenplum Database [4], and Hadoop distributed storage. The HAWQ architecture is different from Greenplum Database because of the underlying distribution storage characteristics and tight integration requirements with Hadoop stack. HAWQ was alpha released in 2012, in production in early 2013, and is now being used by hundreds of Pivotal Hadoop customers. It is SQL compliant and designed to support the large-scale analytics processing for big data [17, 25]. And it has excellent query performance and can also interactively query various data sources in different Hadoop formats.

This paper gives a detailed introduction about the HAWQ architecture and the novel design of various internal components. HAWQ runs side by side with HDFS. A typical HAWQ cluster consists of one master host, one standby master host, two HDFS NameNode hosts with HA configuration, and multiple compute hosts that run both HDFS DataNodes and stateless segments (the basic compute units of HAWQ). User data is stored on HDFS. The master accepts queries through JDBC, ODBC or libpq connections. Queries are parsed, analyzed and optimized into physical query plans. And the dispatcher dispatches plans to segments for execution. The executor executes plans in a pipeline manner and the final results are sent back to the master and presented to clients. On the master, there is a fault detector. When a node fails, segments on the node are marked as "down", and future queries will not be dispatched to these segments anymore. Specifically, we make the following contributions in this paper:

- A stateless segment based database architecture is proposed, which is supported with metadata dispatch and self-described plan execution.
- A UDP based interconnect is designed to eliminate the limitation of TCP for the tuple communication between execution slices.
- A swimming lane based transaction model is proposed to support concurrent update operations. And HDFS is enhanced with truncate operation for transaction rollback.
- Extensive experiments on industry benchmark show that HAWQ is about 40x faster than Stinger, which is reported orders of magnitude faster than the original Hive.

The rest of the paper is organized as follows. We first give an overview of the HAWQ architecture in Section 2. A brief description of query processing is provided in Section 3. In Section 4, the interconnect design is discussed and transaction management is presented in Section 5. The extension framework is discussed in Section 6. Section 7 gives related work. Experimental results on industry benchmark are presented in Section 8. Finally, we conclude the work in Section 9.

2. THE ARCHITECTURE

A HAWQ cluster is built from commodity hardware components, i.e., no proprietary hardware is used. One design choice is to avoid the explicit software customization on special hardware characteristics. This leads to an easy-to-maintain code base, better portability across different platforms and makes the system less sensitive to variations in hardware configurations.

As shown in Figure 1, HAWQ has a layered architecture. An MPP shared-nothing compute layer sits on top of a distributed storage layer. A minimal HAWQ production configuration[1] has three kinds of nodes: HAWQ masters, HDFS NameNodes, and segment hosts running both HAWQ segments and HDFS DataNodes. The master node provides an external interface for the HAWQ cluster. All user interactions go through the master node, which is responsible for authentication and authorization, query parsing, generating query plans, starting query executors and dispatching plans. The master node also tracks the query execution status and gathers the individual results into a final result. The master has a corresponding warm standby master. The standby synchronizes with the master via log shipping. A segment host often has a HDFS DataNode and multiple segments configured to increases the resource utilization of underlying multi-core architecture. HDFS DataNodes are collocated with segments for better data locality. Segments access HDFS through libhdfs3, which is a protobuf based C++ HDFS client. Pivotal Extension Framework (PXF) is an extensible framework, which enables SQL access to a variety of external data sources such as HBase and Accumulo.

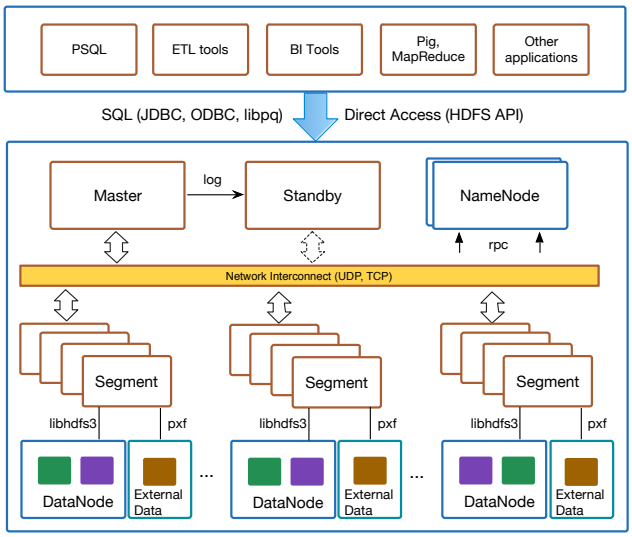


Figure 1: HAWQ architecture

2.1 Interfaces

Like classical relational databases, applications and external tools interact with HAWQ via standard protocols, such as JDBC, ODBC, and libpq that is used by PostgreSQL and Greenplum database. To further simplify the interaction between HAWQ and external systems, especially, new applications developed in Hadoop ecosystem, the design choice here is to enhance standard interfaces with open data format support. External systems can bypass HAWQ, and access directly the HAWQ table files on HDFS. HDFS file system API can be used for this purpose. In addition, open MapReduce InputFormats and OutputFormats for the underlying storage file formats are developed. The feature is quite useful for data loading and data exchange between HAWQ and external applications. For example, MapReduce can directly access table files on HDFS instead of reading HAWQ data through SQL.

2.2 Catalog service

There are two kinds of data: catalog and user data. Catalog is the brain of the system. It describes the system and all of the user

[1]In real deployments, HAWQ can be installed side by side with other components too, for example Hadoop MapReduce.

objects in the system. The catalog is critical to almost all operational processes, such as creating databases and tables, starting and stopping the system, and planning and executing queries. The following are the categories of HAWQ catalog.

- **Database objects**: Description about filespaces, tablespaces, schemas, databases, tables, partitions, columns, indexes, constraints, types, statistics, resource queues, operators, functions and object dependencies.
- **System information**: Information about the segments and segment status.
- **Security**: Users, roles and privileges.
- **Language and encoding**: Languages and encoding supported.

The catalog is stored in the unified catalog service (UCS[2]). External applications can query the catalog using standard SQL. All internal catalog access within HAWQ is provided via a catalog query language, CaQL, which is a subset of SQL. CaQL replaces a more tedious and error-prone process where developers had to manually code catalog queries using C primitives on a combination of internal caches, heaps, indexes, and locking structures, specifying low-level key comparison functions and type conversions. The decision to implement CaQL, instead of using a full functional SQL, was made after a thorough catalog access behavior analysis. Most internal operations are simple OLTP-style lookups using a fixed set of indexes, so sophisticated planning, join optimization, and DDL are unnecessary. A simplified catalog language is faster, and it is much easier to implement and scale from an engineering perspective. In current HAWQ, CaQL only supports basic single-table SELECT, COUNT(), multi-row DELETE, and single row INSERT/UPDATE. More details about the syntax are omitted due to space limitations.

2.3 Data distribution

Besides catalog tables, which are located in UCS, all data is distributed across the cluster, similar to the horizontal partitioning used in Gamma database machine [21]. Several modes are offered to assign data to segments. Each segment has a separate directory on HDFS. Assigning data to segments means storing the data in the segment data directories. The most frequently used policy is hash distribution that distributes rows by hashing of the designated distribution columns. Hash distribution policy gives users the ability to align tables to improve the processing of some important query patterns. For example, if two tables are frequently joined together in queries, and if they are hash distributed with the join key, the equi-join on the two tables can be executed without any data redistribution or shuffling. This improves query performance and saves a lot of network bandwidth. The following syntax is used to designate a set of columns as distribution columns. In this example, the rows of TPC-H $orders$ table are distributed based on hash function against the column $o\_orderkey$. The rows with the same hash value will be distributed to the same segment.

```
CREATE TABLE orders (
  o_orderkey INT8 NOT NULL,
  o_custkey INTEGER NOT NULL
  o_orderstatus CHAR(1) NOT NULL,
  o_totalprice DECIMAL(15,2) NOT NULL,
  o_orderdate DATE NOT NULL,
  o_orderpriority CHAR(15) NOT NULL,
  o_clerk CHAR(15) NOT NULL,
  o_shippriority INTEGER NOT NULL,
  o_comment VARCHAR(79) NOT NULL
) DISTRIBUTED BY (o_orderkey);
```

[2]In current HAWQ release, UCS is located on master only. Future plan is to externalize UCS as a service, which is quite useful for the tight integration and interaction with other Hadoop components.

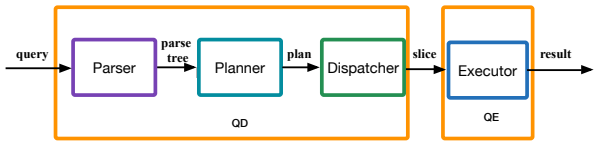


Figure 2: Query execution workflow

HAWQ also provides another distribution policy named RANDOMLY. In random distribution, rows are distributed across segments in a round-robin fashion. Rows with columns having the same values will not necessarily be located on the same segment. A random distribution ensures even data distribution, and it is quite useful for tables with only a very small number of distinct rows. Hashing will assign the data to a small number of segments, thus introduce skew in the data distribution. Random distribution solves this issue quite elegantly.

Table partitioning: Table partitioning is used by HAWQ to support large tables and can also be used to facilitate database maintenance tasks, such as rolling out some old data. Tables are partitioned at CREATE TABLE time using the PARTITION BY clause. Both range partitioning and list partitioning are supported. When you partition a table, you are actually creating a top-level parent table with one or more levels of child tables. Internally, HAWQ creates an inheritance relationship between the top-level table and its underlying partitions. The following example creates a table $sales$ with a date range partition. Each partition of a partitioned table is distributed like a separate table by using the distribution columns.

```
CREATE TABLE sales(
  id INT, date DATE, amt DECIMAL(10,2))
DISTRIBUTED BY (id)
PARTITION BY RANGE (date)(
  START (date '2008-01-01') INCLUSIVE
  END (date '2009-01-01') EXCLUSIVE
  EVERY (INTERVAL '1 month'));
```

Table partitioning is quite helpful for improving query performance, especially for queries that only touch some partitions. Planner considers partition elimination automatically when generating a plan, which saves disk IOs and is faster compared with a full table scan.

2.4 Query execution workflow

The overall query execution workflow is shown in Figure 2. When a user connects to the master, the postmaster forks a query dispatcher (QD) process. The QD, responsible for the user session, interacts with the user and is the controller of the whole query execution process. After receiving a query, it first parses the query into a raw parse tree, and semantic analysis is run against the tree to validate the tree references. Semantic analysis phase accesses the catalog and checks the correctness of tables, columns, and types involved in the query. After semantic analysis, the query parse tree is rewritten according to rewrite rules (for example view definitions).

The planner takes a query parse tree, and generates a parallel execution plan that encodes how the query should be executed. The query plan is then sliced into plan slices at the data movement (mo-

tion) boundary. A slice is an execution unit that does not cross motion boundary. To execute a slice, the QD starts a gang of QEs and dispatches the plan to them. The QEs in the gang execute the same slice against different parts of data, and finally results are assembled and returned to the user through QD.

2.5 Storage

Traditionally, relational data for OLTP systems is stored in rows, i.e., all columns of each tuple are stored together on disk. However, analytical databases tend to have different access patterns than OLTP systems. Instead of having many small transactions with single-row reads and writes, analytical databases often deal with larger and more complex queries that touch much larger volumes of data, i.e., read-mostly with big scanning reads (frequently on selected columns), and infrequent batch appends of data.

HAWQ has built-in flexibility supporting a variety of storage models on HDFS. For each table (or a partition of a table), the user can select the storage and compression settings that suit the way in which the table is accessed. The transformation among different storage models is currently done at the user layer. Automatic storage model transformation is in product roadmap.

- **Row-oriented/Read optimized AO**: Optimized for read-mostly full table scans and bulk append loads. DDL allows optional compression ranging from fast/light to deep/archival schemas such as gzip and quicklz.
- **Column-oriented/Read optimized CO**: Data is vertically partitioned [24], and each column is stored in a series of large, densely packed blocks that can be efficiently compressed (gzip, quicklz and RLE), and it tends to see notably higher compression ratios than row-oriented tables. The implementation scans only those columns required by the query which is good for those workloads suited to column-store. It stores columns in separate segment files.
- **Column-oriented/Parquet**: Similar to CO, data is vertically partitioned. The difference is that Parquet is a PAX [13] like storage format and it stores columns vertically in a row group instead of separate files. Furthermore, it natively supports nested data.

2.6 Fault tolerance

For the high availability of the master node, a standby master instance can be optionally deployed on a separate host. The standby master host serves as a warm standby when the primary master host becomes unavailable. The standby master is kept synchronized by a transaction log replication process, which runs on the standby master host. Since the master does not contain any user data, only the system catalog tables need to be synchronized between the primary and standby. These tables are not updated frequently, but when they are, changes are automatically copied over to the standby master so that it is always kept current with the primary.

From segment side, fault tolerance is different from master mirroring mechanism. HAWQ segments are stateless, which simplifies segment recovery process and provides better availability. A stateless segment means it does not contain private persistent data that should be recovered if it fails. Any alive segment can act as a replacement for a failed segment.

The master has a fault detector that checks the health of all segments periodically. If a failure happens, in-flight queries will fail and the transaction mechanism will keep the consistency of the system. This design is based on the observation that, most of the time, heavy materialization based query recovery is slower than simple query restart. After the failure is detected, the segments on the failed host are marked as "down" in the system catalog. Different user sessions will randomly failover failed segments to the remaining active segments. In a system running concurrent queries, this mechanism provides good load balance across the cluster. After the failed segment host is fixed, a recovery utility can be used to recover the segments on the host, and future queries will be scheduled to the recovered segments. And other utilities are also provided to move failed segments to a new host.

HAWQ supports two level disk failure fault tolerance. User data is stored on HDFS, so disk failures for user data access are masked by HDFS. When one disk failure is identified, HDFS will remove the disk volume from the list of valid disk volumes. There is another kind of disk failures when reading or writing intermediate data for large queries. HAWQ spills intermediate data to local disks for performance reason. For example, when no enough memory is available for a sort operator on a large table, an external sort is used and intermediate data is spilled to local disks. If a failure is encountered during intermediate data access, HAWQ will mark the failed disk as "down" and will not use the failed disk for future queries anymore.

3. QUERY PROCESSING

In this section, we discuss some details about how query processing works. Two critical components in query processing are query planner and executor. The planner accepts a query parse tree and outputs an optimized parallel query plan. A plan describes the execution steps the executor should follow to execute the query. The planner obtains a plan by using a cost-based optimization algorithm to evaluate a vast number of potential plans and selects the one that leads to the most efficient query execution. Based on the data queries access, we classify them into three categories.

- **Master-only queries**: These queries access only catalog tables on master or execute expressions that can be evaluated without dispatching query slices to segments. Master-only queries are handled like queries executed on a single node database.
- **Symmetrically dispatched queries**: Physical query plans are dispatched to all segments, which execute them in parallel against different portions of data. These are the most common queries involving user data distributed on the underlying storage system. According to the complexity of a query, the query plan can be as simple as a scan with a gather motion, and can also contain a lot of intermediate data redistribution operators in order to get a final correct result.
- **Directly dispatched queries**: Whenever the planner can guarantee that a slice accesses data from a single segment directory, the plan is only dispatched to that segment. A typical directly dispatched query is a single value lookup or single row insert query. Direct dispatch saves network bandwidth and improves concurrency of small queries.

In the following sections, we focus on symmetrically dispatched queries which often involve data movement. A query plan contains well-known relational operators, scans, joins, sorts, aggregations, and so on, as well as parallel "motion" operators that describe when and how data should be transferred between nodes. The distribution of each table is tracked in catalog known to query planner at compile time. Based on the knowledge, query planner gets a query plan that takes into account where the data is located. Join and aggregation operations require specific distribution to guarantee correct results. Redistribution often introduces both CPU and network overhead, thus the planner often generates a plan either considering the colocation of data and existing distribution or redistributing correctly the data before other operators. Three kinds of parallel

motion operators are utilized to transfer data among query executors.

- **Broadcast Motion (N:N)**: Every segment sends the input tuples to all other segments. The common usage of broadcast motion is for joins with non equal predicates. One big relation is distributed and the other small relation is broadcasted and thus replicated on each segment.
- **Redistribute Motion (N:N)**: Every segment rehashes the input tuples according to the values of a set of columns and redistributes each tuple to the appropriate segment. It is often used in equi-joins to redistribute non-colocated relations according to the join columns, and aggregation operators to group tuples according to aggregation columns.
- **Gather Motion (N:1)**: Every segment sends the input tuples to a single segment (usually the master). Gather motion is often used by QD to gather final results. It can also be used in the middle of plan before executing an operation that cannot be executed in parallel on different segments.

Encapsulating the distribution logic into self-contained operators enables concise reasoning over query plans [23]. Like other operators, the motion operators are also pipelined, i.e., data is exchanged whenever available. Internally, a software UDP based interconnect is used to implement motion operators, which is described in more details in Section 4.

3.1 Metadata dispatch

As described in Section 2, HAWQ has a rich set of catalog tables. Different sets of metadata are used by different components at different query execution phases. For instance, object definition related catalog tables, like table schemas, database and tablespace information, are used by the parser to do semantic analysis. Statistics are used by the planner to optimize queries, and table, column and function definitions are necessary for query execution.

Catalog exists only on master, and segments are stateless. While executing the slices received from QD, QEs on segments need to access catalog tables. An example usage is: it accesses table schema and column type information to decode the storage format while doing table scan. In order to access catalog, one straightforward method is for QEs to set up connections to the master and query metadata. When the cluster scale is large, there are a large number of QEs querying the master concurrently. The master will become a bottleneck. Thus, "*metadata dispatch*" is proposed to tackle this issue. After a parallel plan is generated, we decorate the plan with the metadata needed in the execution phase. A plan with needed metadata included is called a *self-described plan*. The self-described plan contains all information needed for query execution, so QEs do not need to consult unified catalog service anymore. For some queries, such as insert, some metadata is changed by QEs. To record the changes, we piggyback the changes in the master-to-segment connections. And the changes are updated on master in a batch manner. Typically, metadata change is small, so it does not introduce much overhead to master.

A practical issue for metadata dispatch is the plan size. For a very complex query, the plan size can be multiple megabytes. Here, two optimizations are used to solve the issue. For query execution, a lot of native metadata, such as natively supported types and functions, is read only, and is not changed during the life of the HAWQ. So we bootstrapped a readonly catalog store on each segment that contains all constant metadata. The metadata in the readonly store is excluded from a self-described plan. After the plan is generated, a compression algorithm is run against the plan to further reduce the plan size.

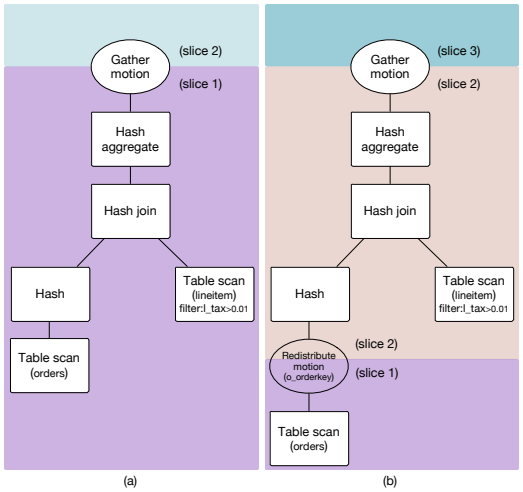


Figure 3: Sliced query plans

3.2 A query processing example

Figure 3 shows the query plans of the following query over standard TPC-H schema with different distributions.

```
SELECT l_orderkey, count(l_quantity)
FROM lineitem, orders
WHERE l_orderkey=o_orderkey AND l_tax>0.01
GROUP BY l_orderkey;
```

In Figure 3(a), $lineitem$ and $orders$ are both distributed by $l\_orderkey$. Since the distribution key and the join key are the same for the two tables, they can be joined locally on each segment without any data movement. It saves both CPU time and network bandwidth. The aggregation in this example is against the same key $l\_orderkey$, so it can be done locally too. Finally, the local segment results are assembled at QD side through a gather motion, and returned to the end user.

From the figure, we can see that there are two slices which is obtained by cutting at the gather motion boundary. The function of a motion is implemented as two parts: a "send" motion and a "receive" motion in the two adjacent slices respectively. The lower slice sends tuples while the upper one receives tuples. During execution, two gangs are created to execute the plan. One gang has the QD that executes the receive part of the gather motion (1-Gang), and the other gang (N-Gang) has one QE for each segment that executes the operators in the bottom slice and sends tuples to QD.

Figure 3(b) shows a different plan, where the $orders$ table is randomly distributed. In order to execute the join, $orders$ is redistributed by hashing the $o\_orderkey$ column. Thus, one more slice is created compared with the plan in Figure 3(a). Query planner makes the decision in a cost based manner. In this example, there are three gangs created to execute the three slices, of which two are N-Gangs and one is 1-Gang. Two N-Gangs communicate to each other by executing the send and receive parts of the redistribution motion respectively.

4. INTERCONNECT

The interconnect is responsible for tuple communication between execution slices (see Figure 4), and it provides communication atomics for motion nodes. Each motion node has many "streams" over which it communicates with its peers. HAWQ has two intercon-

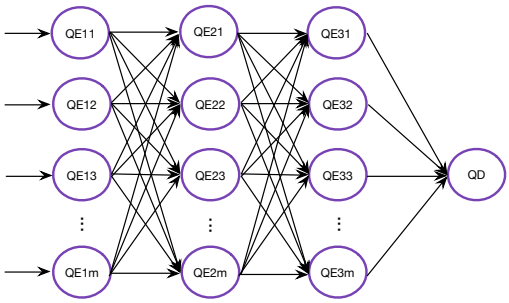


Figure 4: Interconnect for pipelined execution

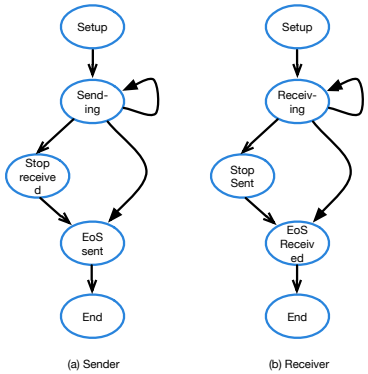


Figure 5: Interconnect sender and receiver state machines

nect implementations based on TCP and UDP respectively. TCP interconnect is limited by the number of slices it can support due to the port number limitation of TCP (about 60k per IP address) and time consuming connection setup step for a large number of concurrent connections. For example, with 1,000 segments, a typical 5-slice query needs to set up about 3 million connections, and on each node, there are thousands of connections. The connectionless UDP based interconnect solves the two issues elegantly by multiplexing multiple tuple-streams over a single UDP socket on each segment and thus provides a better scalability. In this section, we will show the design of the UDP based interconnect, which should achieve the following goals.

- **Reliability**: Network hardware elements and operating system are free to throw out packets, and the operating system UDP protocol does not guarantee reliable packet delivery. The interconnect must be capable of recovering from any packet losses.
- **Ordering**: Operating system UDP protocol does not have packet ordering assurance, so it is the interconnect's responsibility to deliver the packets in order.
- **Flow control**: Receiver processes, operating system and network elements may not match the speed of sender processes, flow control is necessary for UDP based interconnect.
- **Performance and scalability**: The implementation should have high performance and scales without hitting limitation from underlying operating system.
- **Portability**: It should be portable through various platforms and avoid using private features from some hardware and operating systems.

4.1 The protocol

In the interconnect, a packet contains a self-describing header specifying the complete motion node and peer identity along with the session and command-id. Header fields are evenly aligned for performance and portability.

Figure 5 (a) and (b) illustrate the state machines for data sending and data receiving respectively. "EoS" means "End of Stream" which is sent by senders when it has finished tuple sending for the stream, and "Stop" message is used by receivers to stop senders, which is typical for queries with "limit" subclauses when senders have received enough data.

4.2 The implementation

Each sender has a separate "virtual" connection to each receiver. Here we call a connection "virtual" since there is no physical connection for each sender and receiver pair. All the connections in a sender share a common socket. Multiplexing tuple streams from multiple peers over a single UDP socket requires that the kernel socket buffers be emptied rapidly (to avoid packet losses). To facilitate emptying the kernel socket buffers quickly, the communication is multithreaded such that receive, verification, acknowledgment, and receive buffer management execute concurrently with the rest of the executor.

Each connection on sender side has a send queue and an expiration queue ring that is used to store unacknowledged packets and facilitate fast packet expiration check. The expiration queue ring is shared by all sender connections in a sender process. When a sender fills up a packet, it adds the packet to the send queue, and tries to send it if there is capacity for the packet in the receiver side and the current flow control window *cwnd* is not full. The *flow control window* defines the maximal number of packets that have not received acknowledgements from the receiver. When the sender gets an acknowledgement from the receiver, it will free buffers and keep sending packets in the send queue. If the timer for an unacknowledged packet expires, the packet is resent.

At the receiver side, the receiving thread listens on a port and continuously accepts packets from all other senders. For each sender, the receiver has a separate channel to cache the packets from the sender (a separate channel is required to avoid potential deadlock for some types of queries). When the thread gets a packet from a sender, it places it into the corresponding queue and sends an acknowledgement to the sender. The executor thread continuously picks up packets from the receive queues and assembles tuples. Two important fields are contained in an acknowledgement packet: SC (the sequence number of the packet the receiver has already consumed) and SR (the largest packet sequence number it has already received and queued). SC is used to compute the capacity left on the receiver side and SR is used to identify the packets the receiver has already received. The packets that are queued by receiver can be removed from the expiration queue ring in the sender side.

4.3 Flow control

Flow control is used to avoid overwhelming receivers, operating system and intermediate network elements including NICs and switches from sender side. Retransmission timeouts (RTO) are dynamically computed based on round trip time (RTT) which is obtained from the time between the packet is sent and the acknowledgement for the packet is received. A loss based flow control method is implemented to dynamically tune the sending speed. Specifically, a flow control window is used to control the sending speed. Whenever the sender gets an expired packet (which indicates some potential losses that occurred in the operating system and network hardware), it decreases the flow control window to a minimal predefined value. Then a slow start process is used to increase the window to a suitable size.

4.4 Packet losses and out-of-order packets

The interconnect keeps the incoming packets sorted. A naive implementation requires a sort of the incoming packets, and it is obviously not efficient. Here we use a ring buffer to held the packets without any overhead introduced to sort the packets. When a packet loss or out-of-order packet is detected, the receiver immediately sends an OUT-OF-ORDER message to the sender, which contains the packets that were possibly lost. When the sender gets the feedback, it resends the packets. When a duplicate packet is detected in the receiver side, the background thread immediately sends a DUPLICATE message with accumulative acknowledgement information to the sender. When the sender gets the message, it removes any received packets from the expiration queue ring to avoid future duplicated retransmissions.

4.5 Deadlock elimination

One possible deadlock case occurs when there are a lot of lost acknowledgements. It can be explained by an example. Assume a connection has two sender buffers and two receive buffers. Firstly, the sender sends two packets p_1 and p_2 to the receiver. The receiver buffers now are all occupied. If the receiver is very slow at processing packets. It is possible that sender will resend a packet (say p_1), and after receiver gets the packet, it gives sender an acknowledgement that says p_1 and p_2 have been received and buffered. Then the sender removes p_1 and p_2 from expiration queue ring. After the receiver consumes p_1 and p_2, the acknowledgement says that the two packets have been consumed. If the acknowledgement is lost, then sender and receiver will deadlock. The sender does not know the receiver has consumed the packets and will not send or resend packets anymore, and the receiver is waiting for packets. The solution is to introduce a deadlock elimination mechanism. It works as follows. When there is a long period that the sender does not receive any acknowledgement from its peer, and it knows that there is no capacity left on receiver side, the unacknowledged queue is empty and it has some packets to send, it will send a status query message to the receiver. The receiver will respond with an acknowledgement about SC and SR.

5. TRANSACTION MANAGEMENT

HAWQ is fully transactional, which makes it much easier for developers to develop analytic applications. It is because it is very error prone to keep consistency from application side. In this section, we discuss the transaction management and concurrency control in HAWQ. As described earlier, HAWQ distinguishes two kinds of data in HAWQ: catalog and user data. For catalog, write ahead log (WAL) and multi-version concurrency control (MVCC) are used to implement transaction and concurrency control. But for user data, it can be appended, but can not be updated. And HAWQ does not write log and use multiple versions. Visibility is controlled by recording the logical length of data files in system catalog. This works since the HDFS files are append-only. The logical length of a file is the length of useful data in the file. Sometimes, the physical length is not the same as the logical length when an insert transaction fails. And the garbage data needs to be truncated before next write to the file.

Different from many other distributed systems like Greenplum database, HAWQ does not use distributed commit protocol such as two phase commit (2PC). Transaction is only noticeable on master node and segments do not keep transaction states. Self-described plans contain the snapshot information which controls the visibility of each tables. On each segment, dispatched catalog can be modified during the execution, and these modifications are sent back to the master when the query finishes, and finally the master updates all changes in the UCS. Transaction commits only occur on the master node. If a transaction is aborted, the data which has been written into the segment files on HDFS needs to be truncated.

5.1 Isolation levels

Technically, HAWQ uses multi-version concurrency control and supports snapshot isolation [14]. Users can request any one of the four SQL standard isolation levels, but internally, like PostgreSQL, HAWQ only supports: read committed and serializable. Read uncommitted is treated as read committed, while repeatable read is treated as serializable. For serializable isolation level in HAWQ, it does not allow the three phenomena in SQL standard: dirty read, non-repeatable read and phantom read. But the definition is not the same to true mathematical serializability, which does not prevent "write skew" and other phenomena [33] that are in snapshot isolation.

- **Read committed**: A statement can only see rows committed before it began. When you request read uncommitted, actually, you get read committed.
- **Serializable**: All statements of the current transaction can only see rows committed before the first query or data changed in this transaction.

By default, HAWQ sets transactions under "read committed" isolation level. Each query in the transaction gets a snapshot of the database as of the instant the query is started. While the queries in the transaction use the same snapshot at the start of the transaction if the isolation level is set to serializable.

5.2 Locking

HAWQ uses locking techniques to control the conflicts between concurrent DDL and DML statements. For example, the transaction gets an access shared lock before selecting a table, and a concurrent alter table statement will pend on acquirement of an access exclusive lock. The alter table query will continue to execute after the select query is committed or aborted.

HAWQ has its own deadlock checking mechanism. Master node and segments run deadlock checking routine periodically. The transaction is aborted if a deadlock is detected.

5.3 HDFS truncate

Systems with transaction support often need to undo changes made to the underlying storage when a transaction is aborted. Currently HDFS does not support truncate, a standard Posix operation, which is a reverse operation of append. It makes upper layer applications use ugly workarounds, such as keeping track of the discarded byte range per file in a separate metadata store, and periodically running a vacuum process to rewrite compacted files. In Pivotal Hadoop HDFS, we added truncate to support transaction. The signature of the truncate is as follows.

void **truncate**(Path src, long length) throws IOException;

The truncate() function truncates the file to the size which is less or equal to the file length. If the size of the file is smaller than the target length, an IOException is thrown. This is different from Posix truncate semantics. The rationale behind is HDFS does not support overwriting at any position.

For concurrent operations, only single writer/appender/truncater is allowed. Users can only call truncate on closed files. The atomicity of a truncate operation is guaranteed. That is, it succeeds or fails, and it does not leave the file in an undefined state. Concurrent readers may read content of a file that is truncated by a concurrent

truncate operation. But they must be able to read all the data that are not affected by the concurrent truncate operation.

The implementation of truncate is straightforward. At first, a lease is obtained for the to-be-truncated file F. If the truncate operation is at block boundary, the NameNode only needs to delete the to-be-truncated blocks. If the operation is not at block boundary, let the result file be R after truncation, the HDFS client copies the last block B of R to a temporary file T, then the tail blocks of file F including block B, $B+1$ to the end of F are dropped. Then R is obtained by contacting F and T. Finally, the lease for the file is released.

5.4 Concurrent updates

For catalog tables, multi-version concurrency control and locking are used to support concurrent updates. For user data tables, they are append only and stored on HDFS. A light-weight *swimming lane* based method is used to support concurrent inserts. Simply speaking, different inserts change different files, which is quite like different writers are in different *swimming lanes* and do not interfere with each other. For typical OLAP workload, tables are often batch loaded at night time, or continuously updated by a small number of concurrent writers. And files can be appended by another transaction after one transaction is completed. Thus, allocating a separate file for a concurrent writer is reasonable and does not lead to unlimited number of small files. Internally, to support concurrent writers, a catalog table for each user table is created to record the information about all the data files of the table. Each tuple in the catalog table records the logic length, uncompressed length and other useful information of a segment file. The logic length is the most important information to support transaction. The physical file length might differ from the logic length due to failed transactions. Queries use logic length in the transaction snapshot when scanning a file.

6. EXTENSION FRAMEWORK

In this section we describe the PXF, which is a fast and extensible framework that connects HAWQ to any data store of choice. The main idea behind PXF is to provide the ability to use HAWQ's extensive SQL language support, smart query planner, and fast executor to access any kind of data that is external, in the form of a HAWQ external table. Existing PXF connectors include transparent and parallel connectivity to HBase, Accumulo, GemFireXD, Cassandra and Hive tables, various common HDFS file types such as Sequence file, Avro file, RCFile, plain text (delimited, csv, JSON), and various record serialization techniques as well. The framework is highly extensible, and exposes a parallel connector API. It is possible for any user to implement this API in order to create a connector for any kind of data stores, or for a proprietary data format that is used internally as a part of the business process. The result is the ability to use HAWQ to run any type of SQL directly on data sets from such external data stores. This also transparently enables running complex joins between internal HAWQ tables and external PXF tables (e.g, an Avro file on HDFS). And, PXF can export internal HAWQ data into files on HDFS. From transaction support perspective, PXF is not transaction aware.

6.1 Usage

HAWQ and Pivotal Hadoop come with a set of built-in PXF connectors. To connect to a data store that has a built-in connector, we need to specify the connector name and the data source. If a built-in connector does not exist, then PXF can be made to support it as described later on in the Section 6.4. As an example, let us assume we want to access data from an HBase table in parallel, and the HBase table name is 'sales'. And assume that we are interested in column family 'details' and qualifiers 'store', 'price' only. A corresponding HAWQ table with a PXF protocol looks as follows:

```
CREATE EXTERNAL TABLE my_hbase_sales (
  recordkey BYTEA,
  "details:storeid" INT,
  "details:price" DOUBLE)
LOCATION('pxf://<pxf service location>/sales?
profile=HBase')
FORMAT 'CUSTOM' (formatter='pxfwritable_import
');
```

We can then perform all kinds of SQL on these HBase table attributes and HBase row keys. For example,

```
SELECT sum("details:price")
FROM my_hbase_sales
WHERE recordkey < 20130101000000;
```

or even join with an internal HAWQ table

```
SELECT <attrs>
FROM stores s, my_hbase_sales h
WHERE s.name = h."details:storeid";
```

6.2 Benefits

The PXF functionality opens the door to rethinking about restrictive decisions that had to be made previously with respect to placing and processing various data sets. New possibilities are now created as a real connection between various data stores, which can be very easily made on demand, and data can be truly shared. If specific data sets are to be used for various purposes (e.g, operational and analytical), it is now a valid choice to choose to store the data in an in-memory transactional database such as GemFireXD, and directly analyze that same data using HAWQ on demand. Additionally, since interesting joins are made possible, big data designs can be made more flexible. There is also the possibility to insert the data of interests into a local HAWQ table if there is an intention to use this data in many future analytics jobs. Lastly, since HAWQ's planner and executor are highly efficient and proven to be extremely fast, it is the case that many external jobs (such as MapReduce) that can be translated into SQL queries, will run much faster with HAWQ/PXF.

6.3 Advanced functionality

PXF is performance aware, and therefore multiple enhancements are built into the core of the framework. Data locality awareness is implemented in order to assign parallel units of work in a way that will reduce network traffic to a minimum. Each splittable data source, whether an HBase table, an HDFS file, or anything else, consists of one or more "data fragments". For example, in an HDFS file or Hive table, a data fragment can be represented by a file block, and in HBase it can be a table region. The HAWQ query planner uses this information along with the location (host name/ip) of each fragment, and allocates the job of reading that fragment to a segment database that is local to it.

PXF also exposes a filter push down API, which can be used to do record filtering at where the data is stored, instead of reading all the data records and filtering them in HAWQ later on. Also related is the existing support for excluding data partitions that do not match the query filter. This is a classic case for the PXF Hive connector, and it knows which nested data directories to ignore when

processing the query. In both of these cases, HAWQ pushes out the query scan qualifiers and makes it available for the PXF to use as it sees fit.

PXF also supports planner statistics collection. Since the query planner's decisions highly depend on knowledge of the underlying data, PXF has an API to make that happen. It is therefore possible to run an ANALYZE command on a PXF table and store statistics about that table's external data in the HAWQ's catalog, making it available to the query planner when planning a query that involves that PXF table. The statistics collected includes number of tuples, number of pages and some other advanced statistics.

6.4 Building connectors

The connector API includes three plugins that need to be implemented, with another optional one. Once these plugins are implemented they need to be compiled and placed across the system nodes in a way that is accessible by PXF in runtime. The plugin API includes:

- **Fragmenter**: Given a data source location and name, return a list of data fragments and their locations.
- **Accessor**: Given a data fragment, read all the records that belong to that fragment.
- **Resolver**: Given a record from the accessor, deserialize it (if needed) and parse it into attributes. The output of the resolver gets translated into a HAWQ record that can be processed by HAWQ normally.
- **Analyzer** (optional): Given a data source location and name, return a good estimate of the number of records and other statistics of interest.

7. RELATED WORK

A large amount of work has been done in recent years on the distributed data processing area. It can be classified into the following four categories.

- **Data processing framework**: MapReduce [19, 20] and its open source implementation Hadoop [1] become very popular. MapReduce is designed for batch data processing, and delivers a highly-available service on top of a highly-scalable distributed storage. Dryad [26] is a general-purpose distributed data flow execution engine for coarse-grain data-parallel applications. RDDs [40], implemented in Spark [8], is a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.
- **Framework extension**: Pig [30], Hive [36, 37], Tenzing [16] and FlumeJava [15], based on MapReduce, and Dryad-LINQ [39], based on Dryad, are the examples of framework extensions. They provide procedural or declarative query languages to end users. Based on Spark, Shark [38] is a data analysis system running SQL queries and sophisticated analytics functions.
- **Native SQL engines on HDFS**: Like HAWQ, Impala [5], Presto [7] and Drill [3], which is an open source implementation of Dremel [29], adopt database technologies to improve query performance to meet the requirements of ad-hoc queries and provide interactive response time. They all use pipelined execution models. And the major architecture level differences between HAWQ and these systems are at the transaction support (none of them support transactions), the stateless segments and the unique UDP based interconnect.
- **Database and Hadoop mix**: Traditional parallel DBMS vendors such as DB2 [31], Oracle [35], Greenplum database [4], Asterdata [2], Netezza [6], Vertica [11] and Polybase [22] on the other hand, integrate Hadoop with their own database systems to enable their system to access Hadoop data or run MapReduce jobs. HadoopDB [12], which is commercialized in Hadapt [28], builds on MapReduce framework and DBMS technology and provides a hybrid approach to data analysis.

8. EXPERIMENTS

In this section, a thorough experimental study on TPCH benchmark datasets and queries is conducted to evaluate the performance of HAWQ. All the tested systems are deployed on a 20-node cluster connected by a 48-port 10 GigE switch, of which 16 nodes are used for segments and DataNodes, and 4 nodes are used for masters and NameNodes. Each node has two 2.93GHz Intel Xeon 6-core processors, and runs 64-bits CentOS 6.3 with 2.6.23 kernel. And it has 48GB memory, 12×300GB hard disks and one Intel X520 Dual-Port 10 GigE NIC.

8.1 Tested Systems

Stinger[3]**:** Stinger [9] is a community-based effort to optimize Apache Hive. Stinger phase two (available in Hive 0.12) makes several significant improvements, including ORCFile, basic and advanced optimizations, VARCHAR and DATE support, and is reported 35x-45x faster than the original Hive [9]. For the experiments in this paper, MapReduce jobs from Stinger are managed by YARN with data stored on HDFS. A DataNode and a NodeManager are started on each node. We assign 36GB RAM for YARN to use on each node, thus totally 576GB memory on the cluster. The minimum unit of memory to allocate for a container is set to 4GB. To make the performance comparison fair, we tuned the YARN/Stinger default parameters, adapt the test queries, schemas and data formats to leverage the Stinger improvements. Since Stinger cannot run standard TPC-H queries directly, the transformed queries at [10] are used.

HAWQ: On each node, 6 HAWQ segments are configured, which makes 96 segments in total on the cluster. 6GB memory is allocated to each segment. For all other configurations, default parameters are used.

8.2 TPC-H Results

The performance of the two systems are evaluated against TPC-H datasets. The datasets are first generated by the dbgen program as plain text files stored on HDFS, and then loaded into the systems using system-specific storage formats. HAWQ supports three native storage formats: AO, CO and Parquet. And by default, the tables are hash partitioned. For Stinger, ORCFile, the latest and most efficient format in Hive, is used with default configuration. To ensure exclusive access to the cluster's resources, each system executes the benchmark tasks separately.

TPC-H queries are run against datasets of two scales: 160GB (10GB/node) and 1.6TB (100GB/node). Since the 160GB dataset can be loaded into memory completely, this is a CPU-bound case; while the 1.6TB case is IO-bound. We first present the overall query execution time, and then focus on some specific queries to explain how and why HAWQ outperforms Stinger.

8.2.1 Overall Results

Figure 6 outlines the overall execution time of all 22 TPC-H queries against the 160GB datasets. In this task, HAWQ with Parquet storage format run fastest, taking only 172 seconds, followed by CO, 211 seconds, and AO, 239 seconds. Stinger is the slowest

[3]We planned to compare Impala and Presto, but stopped since TPCH queries on large datasets failed frequently on our system due to out of memory issues in the two systems.

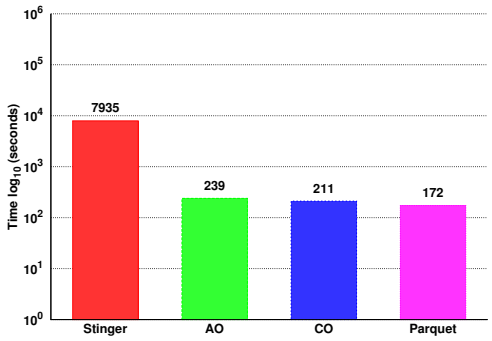


Figure 6: Overall execution time with 160GB

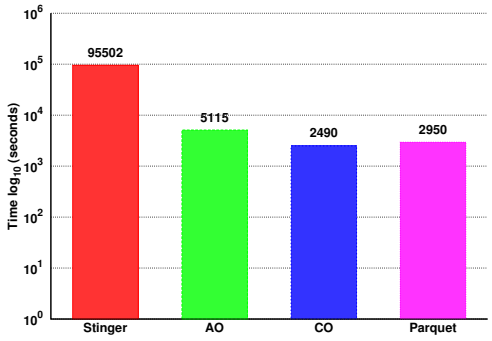


Figure 7: Overall execution time with 1.6TB

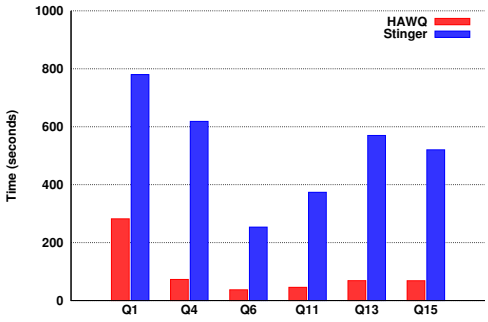


Figure 8: Simple selection queries

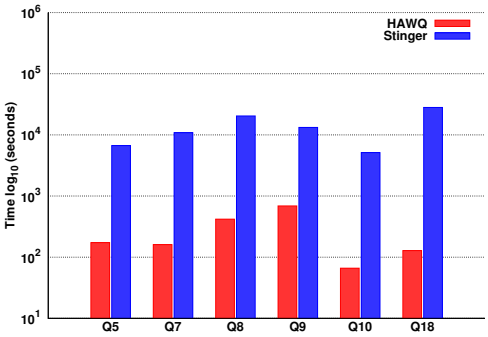


Figure 9: Complex join queries

one, taking more than 7,900 seconds. Thus, no matter with which storage format, HAWQ is about 45x faster than Stinger for this CPU-bound test case.

We only present the overall execution time of 19 out of 22 TPC-H queries on the 1.6TB dataset in Figure 7, since the 3 queries left failed with Stinger (Reducer out of memory). In this task, HAWQ with CO storage format is the fastest one, taking only 2,490 seconds, followed by HAWQ with Parquet, 2,950 seconds, and HAWQ with AO, 5,115 seconds. Hive takes about 96,000 seconds, which is about 40x slower than HAWQ.

The above results are consistent with each other. Both demonstrate that HAWQ is significantly faster than Stinger in either CPU-bound or IO-bound scenarios.

8.2.2 *Specific Query Results*

Based on the complexity of their execution plans, we categorize 12 out of 22 TPC-H queries into two groups: simple selection queries and complex join queries. Due to space limitation, we only consider the 1.6TB dataset here.

Simple selection queries: Six queries fall into this category. All these queries either perform simple selection and aggregation against one single table (Query 1 and 6), or straightforward join of two or three tables (Query 4, 11, 13 and 15). Here we take Query 6 as an example. This query quantities the amount of revenue increase that would have resulted from eliminating certain company-wide discounts in a given percentage range in a given year. Only the table $lineitem$ is involved. The execution plan of this query is simple: first a sequential scan, and then a two-phase aggregation. Figure 8 shows that, for such simple queries, HAWQ is 10x faster than Stinger. There are several factors contributing to this performance gap. Firstly, the task start-up and coordination of HAWQ is much more efficient than YARN. Secondly, data movement in HAWQ is pipelining, while MapReuce jobs materialize the output of each stage, either locally or remotely on HDFS.

Complex join queries: This category contains Query 5, 7, 8, 9, 10, and 18. The common characteristics of these queries are that the selection and aggregation operations are run against several tables of various sizes, involving a join on 3 or more tables (include the two largest tables $orders$ and $lineitem$). These queries are too complex to be optimized manually. For example, Query 5 lists the revenue volume done through local suppliers. Tables $customer$ (240M rows), $orders$ (2.4B rows), $lineitem$ (9.6B rows), $supplier$ (16M rows), $nation$ (25 rows) and $region$ (5 rows) are involved and joined together. In such case, the join order and data movement efficiency have significant impacts on query execution performance. As illustrated in Figure 9, HAWQ is 40x faster than Stinger for complex join queries. Besides the factors discussed previously for simple selection queries, this huge performance gain is also closely related to planning algorithms. Given the table statistics, HAWQ employs a cost-based query optimizing algorithm, which makes it capable to figure out an optimal plan. Stinger uses a simple rule-based algorithm and makes little use of such hints. Thus, most of the time, Stinger can only give a sub-optimal query plan. Furthermore, multiple table join introduces large volume data movement. The HAWQ interconnect can provide much higher data transfer throughput than the HTTP-based one of MapReduce.

8.3 Data Distribution

As we discussed in Section 2.3, HAWQ supports two distribution policies: one is RANDOMLY, and the other one is HASH distribution based on customized distribution columns. The following

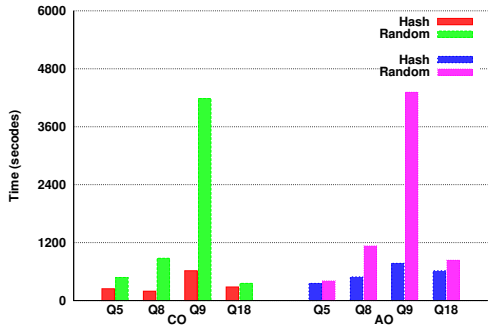
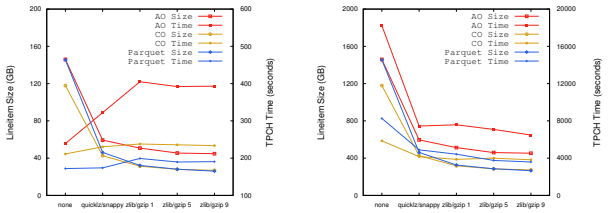


Figure 10: Data Distribution



(a) 160GB dataset. (b) 1.6TB dataset

Figure 11: Compression

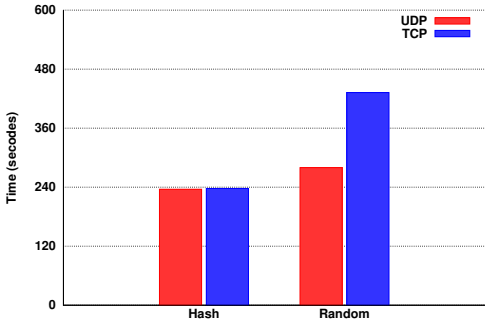
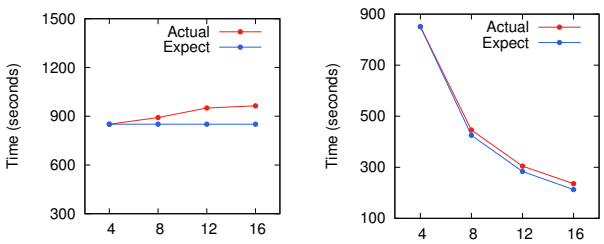


Figure 12: TCP vs. UDP



(a) 40GB per node (b) 160GB per cluster

Figure 13: Scalability

experiment shows how a good designated distribution key can improve query performance.

The results of Query 5, 8, 9 and 18 are presented in Figure 10. Overall, distribution keys bring 2x performance improvement for these queries. Here we take Query 9 as an example to illustrate how this works. Query 9 determines how much profit is made on a given line of parts, broken out by supplier nation and year. This query involves equi-join of tables $part$, $supplier$, $lineitem$, $partsupp$, $orders$, and $nation$, with the join key as the primary key of $part$, $supplier$, $partsupp$ and $nation$ respectively. Designating the primary key as its distribution key for each table saves a lot of data redistribution cost when performing the equi-join, compared to random distribution.

8.4 Compression

We evaluate how different compression algorithms affect the storage usage and query execution time. For AO and CO storage formats, no compression, quicklz, and zlib with level 1, 5, 9 are considered; while for Parquet, we tested no compression, snappy, and gzip with level 1, 5, 9. The compressed data size of the *lineitem* table and TPC-H query execution time under different compression algorithms are shown in Figure 11.

For both datasets, 160GB and 1.6TB, from the perspective of compressed data size, quicklz brings a roughly 3x compression ratio, and zlib with level 1 outperforms quicklz a little, and gets better as its compression level increases, but very slightly. This result suggests that for datasets like TPC-H, zlib with level 1 can bring good enough compression results, and very little benefit is gained by increasing its level. In addition, because CO and Parquet are column-oriented storage formats, better compression results are achieved from them, comparing to AO format, which is row-oriented.

On the other hand, from the execution time view, the two datasets are telling different stories. For the 160GB one, execution time increases as having higher compression ratio, for all three storage formats. This is because doing CPU-intensive compression introduce more extra CPU cost but little IO benefit. The 160GB dataset can be loaded into memory completely with or without compression. For comparison among AO, CO and Parquet, the performance downgrade of AO is more serious than CO and Parquet with compress ratio increased. The reason is column-oriented formats only fetch and decompress columns the queries need, while AO needs to fetch and decompress all the columns. For 1.6TB dataset, the result is opposite. Queries run faster as compression ratio improves. This is because data compression brings some IO benefit, which makes the CPU cost of decompression negligible.

8.5 Interconnect

As discussed in Section 4, HAWQ has two interconnect implementations based on TCP and UDP respectively. In this section, we evaluated the performance of these two interconnect implementations against the 160GB dataset.

Figure 12 shows the overall TPC-H query execution time of two cases: hash distribution and random distribution. UDP and TCP have similar performances for hash distribution. For random distribution, UDP outperforms TCP by 54%. This is because, compared with hash distribution, random distribution leads to deeper plans with more data movements and thus more interconnect connections, and UDP interconnect has less connection setup cost and higher data transfer rate at high concurrency than TCP interconnect.

8.6 Scalability

To evaluate the scalability of HAWQ, two tests are designed. The first one fixes data size per node to 40GB, while the other one fixes the total data size to 160GB and distributes the data to a variable number of nodes by the distribution columns. We run this task on

the cluster size of 4, 8, 12 and 16 nodes. It measures how well HAWQ scales as the cluster size is increased.

The result of the first test is shown in Figure 13(a), where the red ones are the actual test results, while the green ones are the ideal linear expectations. When the cluster size scales from 4 nodes to 16 nodes, the dataset being processed increases from 160GB to 640GB, and the execution time increases slightly, approximately about 13%. This result implies that the size of dataset which HAWQ can process is near-linearly increased as the size of the cluster increases.

Figure 13(b) illustrates the result of the second test. The execution time decreases from 850 seconds to 236 seconds, about 28% of the former one, as the cluster size scales from 4 nodes to 16 nodes. This result suggests that for a fixed dataset, the execution time is near-linearly decreased as the cluster size increases.

9. CONCLUSION

This work presented HAWQ, a massively parallel processing SQL engine that combines the merits of MPP database and Hadoop into a single runtime. On the one hand, employing HDFS as the underlying storage layer, HAWQ provides novel data replication and fault tolerance mechanisms. On the other hand, HAWQ is standard SQL compliant, highly efficient and fully transactional, which are from its inherence of MPP database. The architecture design and implementation are discussed, including query processing, UDP-based interconnect, transaction management, fault tolerance, read optimized storage, and extensible framework for supporting various popular data stores and formats. The experimental study shows that HAWQ outperforms Stinger by a large margin.

10. REFERENCES

[1] Apache hadoop, http://hadoop.apache.org/.

[2] Asterdata, http://www.asterdata.com/sqlh/.

[3] Drill, http://incubator.apache.org/drill/.

[4] Greenplum database, http://www.gopivotal.com.

[5] Impala, https://github.com/cloudera/impala.

[6] Netezza, http://www-01.ibm.com/software/data/netezza/.

[7] Presto, http://prestodb.io.

[8] Spark, http://spark.incubator.apache.org/.

[9] Stinger, http://hortonworks.com/labs/stinger/.

[10] TPC-H on hive, https://github.com/rxin/TPC-H-Hive.

[11] Vertica, http://www.vertica.com.

[12] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.

[13] A. Ailamaki, D. J. Dewitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[14] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[15] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[16] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, and et al. Tenzing: A SQL implementation on the MapReduce framework. In *VLDB*, 2011.

[17] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. In *VLDB*, 2009.

[18] J. C. Corbett, J. Dean, M. Epstein, and et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[20] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[21] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[22] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in Polybase. In *SIGMOD*, 2013.

[23] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, 1990.

[24] S. Harizopoulos, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. In *VLDB*, 2009.

[25] J. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, and et al. The MADlib analytics library or MAD skills, the SQL. In *VLDB*, 2012.

[26] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[27] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of MapReduce: An in-depth study. In *VLDB*, 2010.

[28] D. J. A. Kamil Bajda-Pawlikowski and, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *SIGMOD*, 2011.

[29] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *VLDB*, 2010.

[30] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[31] F. özcan, D. Hoa, K. S. Beyer, A. Balmin, C. J. Liu, and Y. Li. Emerging trends in the enterprise data analytics: connecting Hadoop and DB2 warehouse. In *SIGMOD*, 2011.

[32] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[33] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. In *VLDB*, 2012.

[34] J. Shute, R. Vingralek, B. Samwel, and et al. F1: A distributed SQL database that scales. In *VLDB*, 2013.

[35] X. Su and G. Swart. Oracle in-database Hadoop: When MapReduce meets RDBMS. In *SIGMOD*, 2012.

[36] A. Thusoo, J. S. Sarma, N. Jain, and et al. Hive - a warehousing solution over a MapReduce framework. In *VLDB*, 2009.

[37] A. Thusoo, J. S. Sarma, N. Jain, and et al. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*, 2010.

[38] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, 2013.

[39] Y. Yu, M. Isard, D. Fetterly, and et al. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[40] M. Zaharia, M. Chowdhury, T. Das, and et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.