Impala: A Modern, Open-Source SQL Engine for Hadoop

Marcel Kornacker Alexander Behm Victor Bittorf Taras Bobrovytsky
Casey Ching Alan Choi Justin Erickson Martin Grund Daniel Hecht
Matthew Jacobs Ishaan Joshi Lenni Kuff Dileep Kumar Alex Leblang
Nong Li Ippokratis Pandis Henry Robinson David Rorke Silvius Rus
John Russell Dimitris Tsirogiannis Skye Wanderman-Milne Michael Yoder

Cloudera
http://impala.io/

ABSTRACT

Cloudera Impala is a modern, open-source MPP SQL engine architected from the ground up for the Hadoop data processing environment. Impala provides low latency and high concurrency for BI/analytic read-mostly queries on Hadoop, not delivered by batch frameworks such as Apache Hive. This paper presents Impala from a user's perspective, gives an overview of its architecture and main components and briefly demonstrates its superior performance compared against other popular SQL-on-Hadoop systems.

1. INTRODUCTION

Impala is an open-source [1], fully-integrated, state-of-the-art MPP SQL query engine designed specifically to leverage the flexibility and scalability of Hadoop. Impala's goal is to combine the familiar SQL support and multi-user performance of a traditional analytic database with the scalability and flexibility of Apache Hadoop and the production-grade security and management extensions of Cloudera Enterprise. Impala's beta release was in October 2012 and it GA'ed in May 2013. The most recent version, Impala 2.0, was released in October 2014. Impala's ecosystem momentum continues to accelerate, with nearly one million downloads since its GA.

Unlike other systems (often forks of Postgres), Impala is a brand-new engine, written from the ground up in C++ and Java. It maintains Hadoop's flexibility by utilizing standard components (HDFS, HBase, Metastore, YARN, Sentry) and is able to read the majority of the widely-used file formats (e.g. Parquet, Avro, RCFile). To reduce latency, such as that incurred from utilizing MapReduce or by reading data remotely, Impala implements a distributed architecture based on daemon processes that are responsible for all aspects of query execution and that run on the same machines as the rest of the Hadoop infrastructure. The result is performance that is on par or exceeds that of commercial MPP analytic DBMSs, depending on the particular workload.

This paper discusses the services Impala provides to the user and then presents an overview of its architecture and main components. The highest performance that is achievable today requires using HDFS as the underlying storage manager, and therefore that is the focus on this paper; when there are notable differences in terms of how certain technical aspects are handled in conjunction with HBase, we note that in the text without going into detail.

Impala is the highest performing SQL-on-Hadoop system, especially under multi-user workloads. As Section 7 shows, for single-user queries, Impala is up to 13x faster than alternatives, and *6.7x faster on average*. For multi-user queries, the gap widens: Impala is up to 27.4x faster than alternatives, and *18x faster on average* – or nearly three times faster on average for multi-user queries than for single-user ones.

The remainder of this paper is structured as follows: the next section gives an overview of Impala from the user's perspective and points out how it differs from a traditional RDBMS. Section 3 presents the overall architecture of the system. Section 4 presents the frontend component, which includes a cost-based distributed query optimizer, Section 5 presents the backend component, which is responsible for the query execution and employs runtime code generation, and Section 6 presents the resource/workload management component. Section 7 briefly evaluates the performance of Impala. Section 8 discusses the roadmap ahead and Section 9 concludes.

2. USER VIEW OF IMPALA

Impala is a query engine which is integrated into the Hadoop environment and utilizes a number of standard Hadoop components (Metastore, HDFS, HBase, YARN, Sentry) in order to deliver an RDBMS-like experience. However, there are some important differences that will be brought up in the remainder of this section.

Impala was specifically targeted for integration with standard business intelligence environments, and to that end supports most relevant industry standards: clients can connect via ODBC or JDBC; authentication is accomplished with Kerberos or LDAP; authorization follows the standard SQL roles and privileges [2]. In order to query HDFS-resident

[1] https://github.com/cloudera/impala

[2] This is provided by another standard Hadoop component called Sentry [4], which also makes role-based authorization available to Hive, and other components.

7th Biennial Conference on Innovative Data Systems Research (CIDR'15) January 4-7, 2015, Asilomar, California, USA.

data, the user creates tables via the familiar `CREATE TABLE` statement, which, in addition to providing the logical schema of the data, also indicates the physical layout, such as file format(s) and placement within the HDFS directory structure. Those tables can then be queried with standard SQL syntax.

2.1 Physical schema design

When creating a table, the user can also specify a list of partition columns:

```
CREATE TABLE T (...) PARTITIONED BY (day int, month int) LOCATION '<hdfs-path>' STORED AS PARQUET;
```

For an unpartitioned table, data files are stored by default directly in the root directory [3]. For a partitioned table, data files are placed in subdirectories whose paths reflect the partition columns' values. For example, for day 17, month 2 of table T, all data files would be located in directory `<root>/day=17/month=2/`. Note that this form of partitioning does not imply a collocation of the data of an individual partition: the blocks of the data files of a partition are distributed randomly across HDFS data nodes.

Impala also gives the user a great deal of flexibility when choosing file formats. It currently supports compressed and uncompressed text files, sequence file (a splittable form of text files), RCFile (a legacy columnar format), Avro (a binary row format), and Parquet, the highest-performance storage option (Section 5.3 discusses file formats in more detail). As in the example above, the user indicates the storage format in the `CREATE TABLE` or `ALTER TABLE` statements. It is also possible to select a separate format for each partition individually. For example one can specifically set the file format of a particular partition to Parquet with:

```
ALTER TABLE PARTITION(day=17, month=2) SET FILEFORMAT PARQUET.
```

As an example for when this is useful, consider a table with chronologically recorded data, such as click logs. The data for the current day might come in as CSV files and get converted in bulk to Parquet at the end of each day.

2.2 SQL Support

Impala supports most of the SQL-92 `SELECT` statement syntax, plus additional SQL-2003 analytic functions, and most of the standard scalar data types: integer and floating point types, STRING, CHAR, VARCHAR, TIMESTAMP, and DECIMAL with up to 38 digits of precision. Custom application logic can be incorporated through user-defined functions (UDFs) in Java and C++, and user-defined aggregate functions (UDAs), currently only in C++.

Due to the limitations of HDFS as a storage manager, Impala does not support `UPDATE` or `DELETE`, and essentially only supports bulk insertions (`INSERT INTO ... SELECT ...`) [4]. Unlike in a traditional RDBMS, the user can add data to a table simply by copying/moving data files into the directory location of that table, using HDFS's API. Alternatively, the same can be accomplished with the `LOAD DATA` statement.

Similarly to bulk insert, Impala supports bulk data deletion by dropping a table partition (`ALTER TABLE DROP PARTITION`). Because it is not possible to update HDFS files in-place, Impala does not support an `UPDATE` statement. Instead, the user typically recomputes parts of the data set to incorporate updates, and then replaces the corresponding data files, often by dropping and re-adding the partition

After the initial data load, or whenever a significant fraction of the table's data changes, the user should run the `COMPUTE STATS <table>` statement, which instructs Impala to gather statistics on the table. Those statistics will subsequently be used during query optimization.

3. ARCHITECTURE

Impala is a massively-parallel query execution engine, which runs on hundreds of machines in existing Hadoop clusters. It is decoupled from the underlying storage engine, unlike traditional relational database management systems where the query processing and the underlying storage engine are components of a single tightly-coupled system. Impala's high-level architecture is shown in Figure 1.

An Impala deployment is comprised of three services. The Impala daemon (impalad) service is dually responsible for accepting queries from client processes and orchestrating their execution across the cluster, and for executing individual query *fragments* on behalf of other Impala daemons. When an Impala daemon operates in the first role by managing query execution, it is said to be the *coordinator* for that query. However, all Impala daemons are symmetric; they may all operate in all roles. This property helps with fault-tolerance, and with load-balancing.

One Impala daemon is deployed on every machine in the cluster that is also running a *datanode* process - the block server for the underlying HDFS deployment - and therefore there is typically one Impala daemon on every machine. This allows Impala to take advantage of data locality, and to read blocks from the filesystem without having to use the network.

The Statestore daemon (statestored) is Impala's metadata publish-subscribe service, which disseminates cluster-wide metadata to all Impala processes. There is a single statestored instance, which is described in more detail in Section 3.1 below.

Finally, the Catalog daemon (catalogd), described in Section 3.2, serves as Impala's catalog repository and metadata access gateway. Through the catalogd, Impala daemons may execute DDL commands that are reflected in external catalog stores such as the Hive Metastore. Changes to the system catalog are broadcast via the statestore.

All these Impala services, as well as several configuration options, such as the sizes of the resource pools, the available memory, etc.. (see Section 6 for more details about resource and workload management) are also exposed to Cloudera Manager, a sophisticated cluster management application[5]. Cloudera Manager can administer not only Impala but also pretty much every service for a holistic view of a Hadoop deployment.

[3] However, all data files that are located in any directory below the root are part of the table's data set. That is a common approach for dealing with unpartitioned tables, employed also by Apache Hive.

[4] We should also note that Impala supports the `VALUES` clause. However, for HDFS-backed tables this will generate one file per `INSERT` statement, which leads to very poor performance for most applications. For HBase-backed tables, the `VALUES` variant performs single-row inserts by means of the HBase API.

[5] http://www.cloudera.com/content/cloudera/en/products-and-services/cloudera-enterprise/cloudera-manager.html

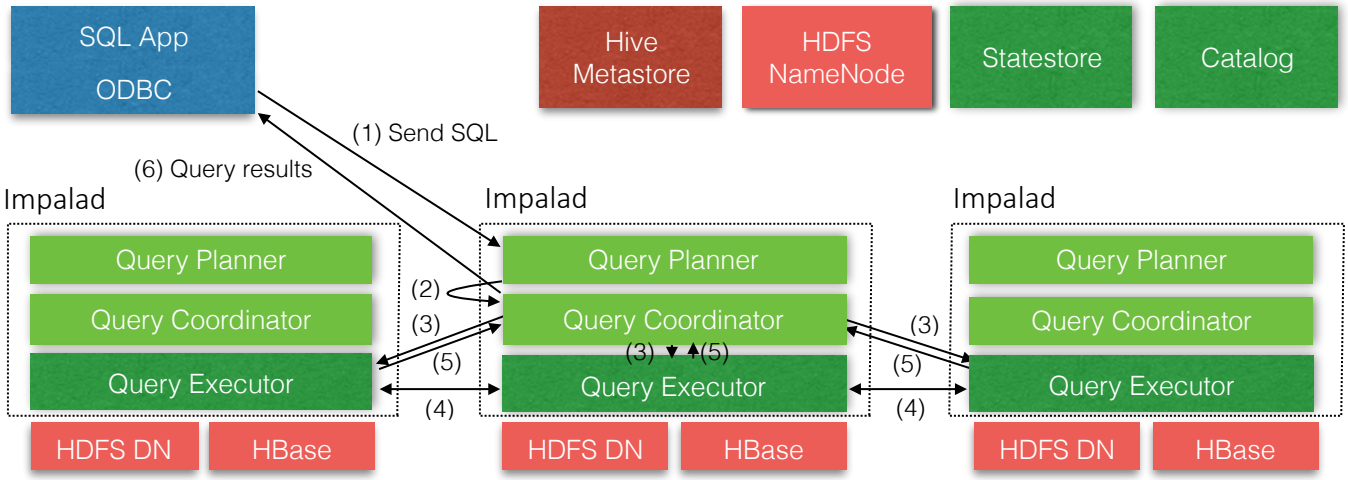

Figure 1: Impala is a distributed query processing system for the Hadoop ecosystem. This figure also shows the flow during query processing.

3.1 State distribution

A major challenge in the design of an MPP database that is intended to run on hundreds of nodes is the coordination and synchronization of cluster-wide metadata. Impala's symmetric-node architecture requires that all nodes must be able to accept and execute queries. Therefore all nodes must have, for example, up-to-date versions of the system catalog and a recent view of the Impala cluster's membership so that queries may be scheduled correctly.

We might approach this problem by deploying a separate cluster-management service, with ground-truth versions of all cluster-wide metadata. Impala daemons could then query this store lazily (i.e. only when needed), which would ensure that all queries were given up-to-date responses. However, a fundamental tenet in Impala's design has been to avoid synchronous RPCs wherever possible on the critical path of any query. Without paying close attention to these costs, we have found that query latency is often compromised by the time taken to establish a TCP connection, or load on some remote service. Instead, we have designed Impala to *push* updates to all interested parties, and have designed a simple publish-subscribe service called the *statestore* to disseminate metadata changes to a set of subscribers.

The statestore maintains a set of topics, which are arrays of (key, value, version) triplets called *entries* where 'key' and 'value' are byte arrays, and 'version' is a 64-bit integer. A topic is defined by an application, and so the statestore has no understanding of the contents of any topic entry. Topics are persistent through the lifetime of the statestore, but are not persisted across service restarts. Processes that wish to receive updates to any topic are called *subscribers*, and express their interest by registering with the statestore at start-up and providing a list of topics. The statestore responds to registration by sending the subscriber an initial topic update for each registered topic, which consists of all the entries currently in that topic.

After registration, the statestore periodically sends two kinds of messages to each subscriber. The first kind of message is a topic update, and consists of all changes to a topic (new entries, modified entries and deletions) since the last update was successfully sent to the subscriber. Each subscriber maintains a per-topic most-recent-version identifier which allows the statestore to only send the delta between updates. In response to a topic update, each subscriber sends a list of changes it wishes to make to its subscribed topics. Those changes are guaranteed to have been applied by the time the next update is received.

The second kind of statestore message is a *keepalive*. The statestore uses keepalive messages to maintain the connection to each subscriber, which would otherwise time-out its subscription and attempt to re-register. Previous versions of the statestore used topic update messages for both purposes, but as the size of topic updates grew it became difficult to ensure timely delivery of updates to each subscriber, leading to false-positives in the subscriber's failure-detection process.

If the statestore detects a failed subscriber (for example, by repeated failed keepalive deliveries), it will cease sending updates. Some topic entries may be marked as 'transient', meaning that if their 'owning' subscriber should fail, they will be removed. This is a natural primitive with which to maintain liveness information for the cluster in a dedicated topic, as well as per-node load statistics.

The statestore provides very weak semantics: subscribers may be updated at different rates (although the statestore tries to distribute topic updates fairly), and may therefore have very different views of the content of a topic. However, Impala only uses topic metadata to make decisions locally, without any coordination across the cluster. For example, query planning is performed on a single node based on the catalog metadata topic, and once a full plan has been computed, all information required to execute that plan is distributed directly to the executing nodes. There is no requirement that an executing node should know about the same version of the catalog metadata topic.

Although there is only a single statestore process in existing Impala deployments, we have found that it scales well to medium sized clusters and, with some configuration, can serve our largest deployments. The statestore does not persist any metadata to disk: all current metadata is pushed to the statestore by live subscribers (e.g. load information).

Therefore, should a statestore restart, its state can be recovered during the initial subscriber registration phase. Or if the machine that the statestore is running on fails, a new statestore process can be started elsewhere, and subscribers may fail over to it. There is no built-in failover mechanism in Impala, instead deployments commonly use a retargetable DNS entry to force subscribers to automatically move to the new process instance.

3.2 Catalog service

Impala's *catalog service* serves catalog metadata to Impala daemons via the statestore broadcast mechanism, and executes DDL operations on behalf of Impala daemons. The catalog service pulls information from third-party metadata stores (for example, the Hive Metastore or the HDFS Namenode), and aggregates that information into an Impala-compatible catalog structure. This architecture allows Impala to be relatively agnostic of the metadata stores for the storage engines it relies upon, which allows us to add new metadata stores to Impala relatively quickly (e.g. HBase support). Any changes to the system catalog (e.g. when a new table has been loaded) are disseminated via the statestore.

The catalog service also allows us to augment the system catalog with Impala-specific information. For example, we register user-defined-functions only with the catalog service (without replicating this to the Hive Metastore, for example), since they are specific to Impala.

Since catalogs are often very large, and access to tables is rarely uniform, the catalog service only loads a skeleton entry for each table it discovers on startup. More detailed table metadata can be loaded lazily in the background from its third-party stores. If a table is required before it has been fully loaded, an Impala daemon will detect this and issue a prioritization request to the catalog service. This request blocks until the table is fully loaded.

4. FRONTEND

The Impala frontend is responsible for compiling SQL text into query plans executable by the Impala backends. It is written in Java and consists of a fully-featured SQL parser and cost-based query optimizer, all implemented from scratch. In addition to the basic SQL features (select, project, join, group by, order by, limit), Impala supports inline views, uncorrelated and correlated subqueries (that are rewritten as joins), all variants of outer joins as well as explicit left/right semi- and anti-joins, and analytic window functions.

The query compilation process follows a traditional division of labor: Query parsing, semantic analysis, and query planning/optimization. We will focus on the latter, most challenging, part of query compilation. The Impala query planner is given as input a parse tree together with query-global information assembled during semantic analysis (table/column identifiers, equivalence classes, etc.). An executable query plan is constructed in two phases: (1) Single node planning and (2) plan parallelization and fragmentation.

In the first phase, the parse tree is translated into a non-executable single-node plan tree, consisting of the following plan nodes: HDFS/HBase scan, hash join, cross join, union, hash aggregation, sort, top-n, and analytic evaluation. This step is responsible for assigning predicates at the lowest possible plan node, inferring predicates based on equivalence classes, pruning table partitions, setting limits/offsets, applying column projections, as well as performing some cost-based plan optimizations such as ordering and coalescing analytic window functions and join reordering to minimize the total evaluation cost. Cost estimation is based on table/partition cardinalities plus distinct value counts for each column [6]; histograms are currently not part of the statistics. Impala uses simple heuristics to avoid exhaustively enumerating and costing the entire join-order space in common cases.

The second planning phase takes the single-node plan as input and produces a distributed execution plan. The general goal is to minimize data movement and maximize scan locality: in HDFS, remote reads are considerably slower than local ones. The plan is made distributed by adding exchange nodes between plan nodes as necessary, and by adding extra non-exchange plan nodes to minimize data movement across the network (e.g., local aggregation nodes). During this second phase, we decide the join strategy for every join node (the join order is fixed at this point). The supported join strategies are broadcast and partitioned. The former replicates the entire build side of a join to all cluster machines executing the probe, and the latter hash-redistributes both the build and probe side on the join expressions. Impala chooses whichever strategy is estimated to minimize the amount of data exchanged over the network, also exploiting existing data partitioning of the join inputs.

All aggregation is currently executed as a local pre-aggregation followed by a merge aggregation operation. For grouping aggregations, the pre-aggregation output is partitioned on the grouping expressions and the merge aggregation is done in parallel on all participating nodes. For non-grouping aggregations, the merge aggregation is done on a single node. Sort and top-n are parallelized in a similar fashion: a distributed local sort/top-n is followed by a single-node merge operation. Analytic expression evaluation is parallelized based on the partition-by expressions. It relies on its input being sorted on the partition-by/order-by expressions. Finally, the distributed plan tree is split up at exchange boundaries. Each such portion of the plan is placed inside a plan fragment, Impala's unit of backend execution. A plan fragment encapsulates a portion of the plan tree that operates on the same data partition on a single machine.

Figure 2 illustrates in an example the two phases of query planning. The left side of the figure shows the single-node plan of a query joining two HDFS tables (`t1, t2`) and one HBase table (`t3`) followed by an aggregation and order by with limit (top-n). The right-hand side shows the distributed, fragmented plan. Rounded rectangles indicate fragment boundaries and arrows data exchanges. Tables `t1` and `t2` are joined via the partitioned strategy. The scans are in a fragment of their own since their results are immediately exchanged to a consumer (the join node) which operates on a hash-based partition of the data, whereas the table data is randomly partitioned. The following join with `t3` is a broadcast join placed in the same fragment as the join between `t1` and `t2` because a broadcast join preserves the existing data partition (the results of joining `t1`, `t2`, and `t3` are still hash partitioned based on the join keys of `t1` and `t2`). After the joins we perform a two-phase distributed aggregation, where a pre-aggregation is computed in the same fragment as the last join. The pre-aggregation results are hash-exchanged based on the grouping keys, and then

[6] We use the HyperLogLog algorithm [5] for distinct value estimation.

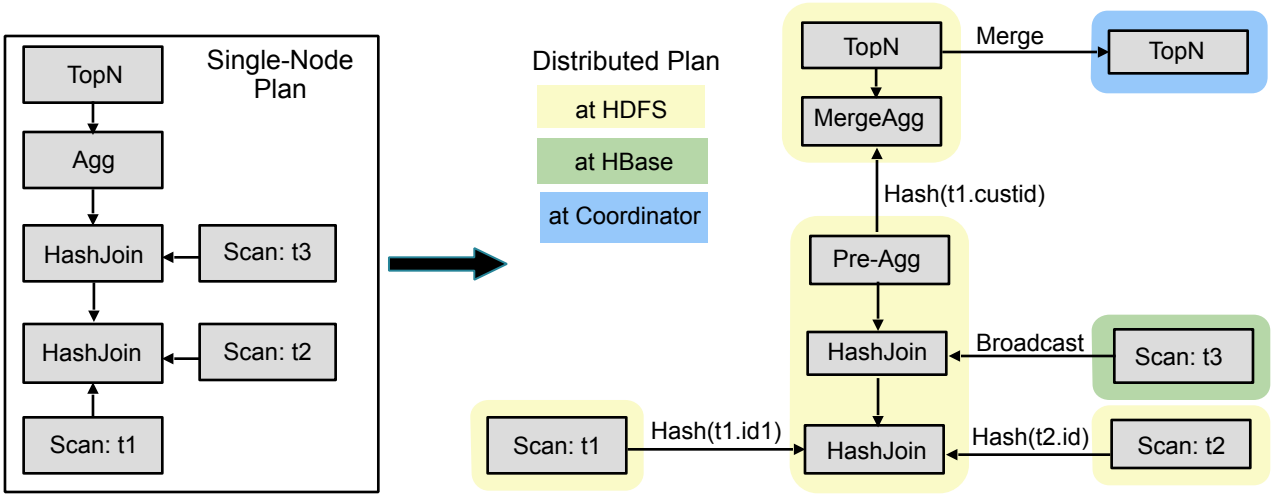

Figure 2: Example of the two phase query optimization.

aggregated once more to compute the final aggregation result. The same two-phased approach is applied to the top-n, and the final top-n step is performed at the coordinator, which returns the results to the user.

5. BACKEND

Impala's backend receives query fragments from the frontend and is responsible for their fast execution. It is designed to take advantage of modern hardware. The backend is written in C++ and uses code generation at runtime to produce efficient codepaths (with respect to instruction count) and small memory overhead, especially compared to other engines implemented in Java.

Impala leverages decades of research in parallel databases. The execution model is the traditional Volcano-style with Exchange operators [7]. Processing is performed batch-at-a-time: each `GetNext()` call operates over batches of rows, similar to [10]. With the exception of "stop-and-go" operators (e.g. sorting), the execution is fully pipeline-able, which minimizes the memory consumption for storing intermediate results. When processed in memory, the tuples have a canonical in-memory row-oriented format.

Operators that may need to consume lots of memory are designed to be able to spill parts of their working set to disk if needed. The operators that are spillable are the hash join, (hash-based) aggregation, sorting, and analytic function evaluation.

Impala employs a partitioning approach for the hash join and aggregation operators. That is, some bits of the hash value of each tuple determine the target partition and the remaining bits for the hash table probe. During normal operation, when all hash tables fit in memory, the overhead of the partitioning step is minimal, within 10% of the performance of a non-spillable non-partitioning-based implementation. When there is memory-pressure, a "victim" partition may be spilled to disk, thereby freeing memory for other partitions to complete their processing. When building the hash tables for the hash joins and there is reduction in cardinality of the build-side relation, we construct a Bloom filter which is then passed on to the probe side scanner, implementing a simple version of a semi-join.

```
IntVal my_func(const IntVal& v1, const IntVal& v2) {
  return IntVal(v1.val * 7 / v2.val);
}
```

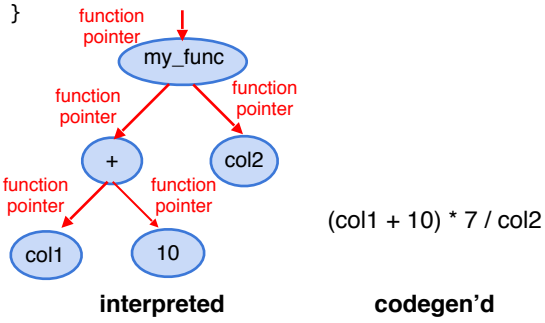

Figure 3: Interpreted vs codegen'ed code in Impala.

5.1 Runtime Code Generation

Runtime code generation using LLVM [8] is one of the techniques employed extensively by Impala's backend to improve execution times. Performance gains of 5x or more are typical for representative workloads.

LLVM is a compiler library and collection of related tools. Unlike traditional compilers that are implemented as standalone applications, LLVM is designed to be modular and reusable. It allows applications like Impala to perform just-in-time (JIT) compilation within a running process, with the full benefits of a modern optimizer and the ability to generate machine code for a number of architectures, by exposing separate APIs for all steps of the compilation process.

Impala uses runtime code generation to produce query-specific versions of functions that are critical to performance. In particular, code generation is applied to "inner loop" functions, i.e., those that are executed many times (for every tuple) in a given query, and thus constitute a large portion of the total time the query takes to execute. For example, a function used to parse a record in a data file into Impala's in-memory tuple format must be called for every record in every data file scanned. For queries scanning large tables, this could be billions of records or more. This function must

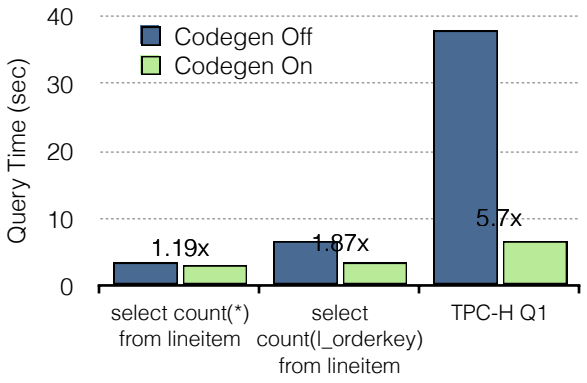

Figure 4: Impact in performance of run-time code generation in Impala.

therefore be extremely efficient for good query performance, and even removing a few instructions from the function's execution can result in large query speedups.

Without code generation, inefficiencies in function execution are almost always necessary in order to handle runtime information not known at program compile time. For example, a record-parsing function that only handles integer types will be faster at parsing an integer-only file than a function that handles other data types such as strings and floating-point numbers as well. However, the schemas of the files to be scanned are unknown at compile time, and so a general-purpose function must be used, even if at runtime it is known that more limited functionality is sufficient.

A source of large runtime overheads are virtual functions. Virtual function calls incur a large performance penalty, particularly when the called function is very simple, as the calls cannot be inlined. If the type of the object instance is known at runtime, we can use code generation to replace the virtual function call with a call directly to the correct function, which can then be inlined. This is especially valuable when evaluating expression trees. In Impala (as in many systems), expressions are composed of a tree of individual operators and functions, as illustrated in the left-hand side of Figure Figure 3. Each type of expression that can appear in a tree is implemented by overriding a virtual function in the expression base class, which recursively calls its child expressions. Many of these expression functions are quite simple, e.g., adding two numbers. Thus, the cost of calling the virtual function often far exceeds the cost of actually evaluating the function. As illustrated in Figure 3, by resolving the virtual function calls with code generation and then inlining the resulting function calls, the expression tree can be evaluated directly with no function call overhead. In addition, inlining functions increases instruction-level parallelism, and allows the compiler to make further optimizations such as subexpression elimination across expressions.

Overall, JIT compilation has an effect similar to custom-coding a query. For example, it eliminates branches, unrolls loops, propagates constants, offsets and pointers, inlines functions. Code generation has a dramatic impact on performance, as shown in Figure 4. For example, in a 10-node cluster with each node having 8 cores, 48GB RAM and 12 disks, we measure the impact of codegen. We are using an Avro TPC-H database of scaling factor 100 and we run simple aggregation queries. Code generation speeds up the execution by up to 5.7x, with the speedup increasing with the query complexity.

5.2 I/O Management

Efficiently retrieving data from HDFS is a challenge for all SQL-on-Hadoop systems. In order to perform data scans from both disk and memory at or near hardware speed, Impala uses an HDFS feature called *short-circuit local reads* [3] to bypass the DataNode protocol when reading from local disk. Impala can read at almost disk bandwidth (approx. 100MB/s per disk) and is typically able to saturate all available disks. We have measured that with 12 disks, Impala is capable of sustaining I/O at 1.2GB/sec. Furthermore, *HDFS caching* [2] allows Impala to access memory-resident data at memory bus speed and also saves CPU cycles as there is no need to copy data blocks and/or checksum them.

Reading/writing data from/to storage devices is the responsibility of the I/O manager component. The I/O manager assigns a fixed number of worker threads per physical disk (one thread per rotational disk and eight per SSD), providing an asynchronous interface to clients (e.g. scanner threads). The effectiveness of Impala's I/O manager was recently corroborated by [6], which shows that Impala's read throughput is from 4x up to 8x higher than the other tested systems.

5.3 Storage Formats

Impala supports most popular file formats: Avro, RC, Sequence, plain text, and Parquet. These formats can be combined with different compression algorithms, such as snappy, gzip, bz2.

In most use cases we recommend using Apache Parquet, a state-of-the-art, open-source columnar file format offering both high compression and high scan efficiency. It was co-developed by Twitter and Cloudera with contributions from Criteo, Stripe, Berkeley AMPlab, and LinkedIn. In addition to Impala, most Hadoop-based processing frameworks including Hive, Pig, MapReduce and Cascading are able to process Parquet.

Simply described, Parquet is a customizable PAX-like [1] format optimized for large data blocks (tens, hundreds, thousands of megabytes) with built-in support for nested data. Inspired by Dremel's ColumnIO format [9], Parquet stores nested fields column-wise and augments them with minimal information to enable re-assembly of the nesting structure from column data at scan time. Parquet has an extensible set of column encodings. Version 1.2 supports run-length and dictionary encodings and version 2.0 added support for delta and optimized string encodings. The most recent version (Parquet 2.0) also implements embedded statistics: inlined column statistics for further optimization of scan efficiency, e.g. min/max indexes.

As mentioned earlier, Parquet offers both high compression and scan efficiency. Figure 5 (left) compares the size on disk of the Lineitem table of a TPC-H database of scaling factor 1,000 when stored in some popular combinations of file formats and compression algorithms. Parquet with snappy compression achieves the best compression among them. Similarly, Figure 5 (right) shows the Impala execution times for various queries from the TPC-DS benchmark when the database is stored in plain text, Sequence, RC, and Parquet formats. Parquet consistently outperforms by up to 5x all the other formats.

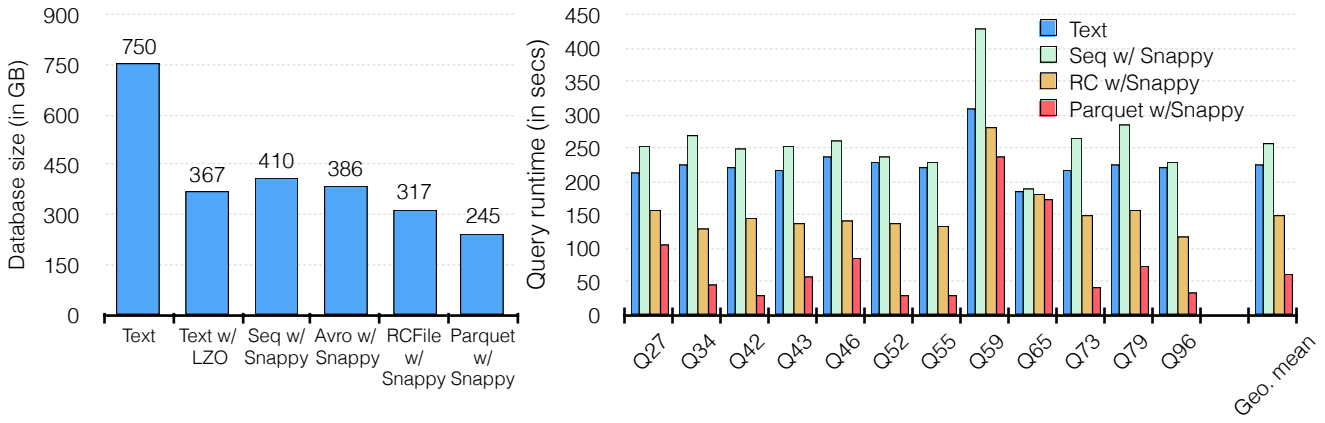

Figure 5: (Left) Comparison of the compression ratio of popular combinations of file formats and compression. (Right) Comparison of the query efficiency of plain text, SEQUENCE, RC, and Parquet in Impala.

6. RESOURCE/WORKLOAD MANAGEMENT

One of the main challenges for any cluster framework is careful control of resource consumption. Impala often runs in the context of a busy cluster, where MapReduce tasks, ingest jobs and bespoke frameworks compete for finite CPU, memory and network resources. The difficulty is to coordinate resource scheduling between queries, and perhaps between frameworks, without compromising query latency or throughput.

Apache YARN [12] is the current standard for resource mediation on Hadoop clusters, which allows frameworks to share resources such as CPU and memory without partitioning the cluster. YARN has a centralized architecture, where frameworks make requests for CPU and memory resources which are arbitrated by the central *Resource Manager* service. This architecture has the advantage of allowing decisions to be made with full knowledge of the cluster state, but it also imposes a significant latency on resource acquisition. As Impala targets workloads of many thousands of queries per second, we found the resource request and response cycle to be prohibitively long.

Our approach to this problem was two-fold: first, we implemented a complementary but independent admission control mechanism that allowed users to control their workloads without costly centralized decision-making. Second, we designed an intermediary service to sit between Impala and YARN with the intention of correcting some of the impedance mismatch. This service, called *Llama* for Low-Latency Application MAster, implements resource caching, gang scheduling and incremental allocation changes while still deferring the actual scheduling decisions to YARN for resource requests that don't hit Llama's cache.

The rest of this section describes both approaches to resource management with Impala. Our long-term goal is to support mixed-workload resource management through a single mechanism that supports both the low latency decision making of admission control, and the cross-framework support of YARN.

6.1 Llama and YARN

Llama is a standalone daemon to which all Impala daemons send per-query resource requests. Each resource request is associated with a resource pool, which defines the fair share of the the cluster's available resources that a query may use.

If resources for the resource pool are available in Llama's resource cache, Llama returns them to the query immediately. This fast path allows Llama to circumvent YARN's resource allocation algorithm when contention for resources is low. Otherwise, Llama forwards the request to YARN's resource manager, and waits for all resources to be returned. This is different from YARN's 'drip-feed' allocation model where resources are returned as they are allocated. Impala's pipelined execution model requires all resources to be available simultaneously so that all query fragments may proceed in parallel.

Since resource estimations for query plans, particularly over very large data sets, are often inaccurate, we allow Impala queries to adjust their resource consumption estimates during execution. This mode is not supported by YARN, instead we have Llama issue new resource requests to YARN (e.g. asking for 1GB more of memory per node) and then aggregate them into a single resource allocation from Impala's perspective. This *adapter* architecture has allowed Impala to fully integrate with YARN without itself absorbing the complexities of dealing with an unsuitable programming interface.

6.2 Admission Control

In addition to integrating with YARN for cluster-wide resource management, Impala has a built-in admission control mechanism to throttle incoming requests. Requests are assigned to a resource pool and admitted, queued, or rejected based on a policy that defines per-pool limits on the maximum number of concurrent requests and the maximum memory usage of requests. The admission controller was designed to be fast and decentralized, so that incoming requests to any Impala daemon can be admitted without making synchronous requests to a central server. State required to make admission decisions is disseminated among Impala daemons via the statestore, so every Impala daemon is able to make admission decisions based on its aggregate view of the global state without any additional synchronous communication on the request execution path. However, because shared state is received asynchronously, Impala daemons may make decisions locally that result in exceeding limits specified by

the policy. In practice this has not been problematic because state is typically updated faster than non-trivial queries. Further, the admission control mechanism is designed primarily to be a simple throttling mechanism rather than a resource management solution such as YARN.

Resource pools are defined hierarchically. Incoming requests are assigned to a resource pool based on a placement policy and access to pools can be controlled using ACLs. The configuration is specified with a YARN fair scheduler allocation file and Llama configuration, and Cloudera Manager provides a simple user interface to configure resource pools, which can be modified without restarting any running services.

7. EVALUATION

The purpose of this section is not to exhaustively evaluate the performance of Impala, but mostly to give some indications. There are independent academic studies that have derived similar conclusions, e.g. [6].

7.1 Experimental setup

All the experiments were run on the same 21-node cluster. Each node in the cluster is a 2-socket machine with 6-core Intel Xeon CPU E5-2630L at 2.00GHz. Each node has 64GB RAM and 12 932GB disk drives (one for the OS, the rest for HDFS).

We run a decision-support style benchmark consisting of a subset of the queries of TPC-DS on a 15TB scale factor data set. In the results below we categorize the queries based on the amount of data they access, into *interactive*, *reporting*, and *deep analytic* queries. In particular, the interactive bucket contains queries: q19, q42, q52, q55, q63, q68, q73, and q98; the reporting bucket contains queries: q27, q3, q43, q53, q7, and q89; and the deep analytic bucket contains queries: q34, q46, q59, q79, and ss_max. The kit we use for these measurements is publicly available [7].

For our comparisons we used the most popular SQL-on-Hadoop systems for which we were able to show results [8]: Impala, Presto, Shark, SparkSQL, and Hive 0.13. Due to the lack of a cost-based optimizer in all tested engines except Impala we tested all engines with queries that had been converted to SQL-92 style joins. For consistency, we ran those same queries against Impala, although Impala produces identical results without these modifications.

Each engine was assessed on the file format that it performs best on, while consistently using Snappy compression to ensure fair comparisons: Impala on Apache Parquet, Hive 0.13 on ORC, Presto on RCFile, and SparkSQL on Parquet.

7.2 Single User Performance

Figure 6 compares the performance of the four systems on single-user runs, where a single user is repeatedly submitting queries with zero think time. Impala outperforms all alternatives on single-user workloads across all queries run. Impala's performance advantage ranges from 2.1x to 13.0x and on average is 6.7x faster. Actually, this is a wider gap of

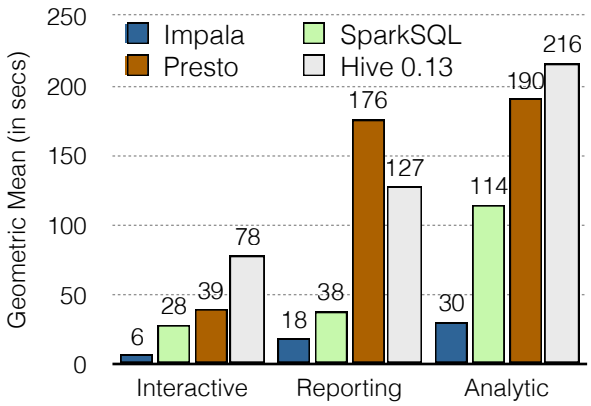

Figure 6: Comparison of query response times on single-user runs.

performance advantage against Hive 0.13 (from an average of 4.9x to 9x) and Presto (from an average of 5.3x to 7.5x) from earlier versions of Impala [9].

7.3 Mutli-User Performance

Impala's superior performance becomes more pronounced in multi-user workloads, which are ubiquitous in real-world applications. Figure 7 (left) shows the response time of the four systems when there are 10 concurrent users submitting queries from the interactive category. In this scenario, Impala outperforms the other systems from 6.7x to 18.7x when going from single user to concurrent user workloads. The speedup varies from 10.6x to 27.4x depending on the comparison. Note that Impala's speed under 10-user load was nearly half that under single-user load–whereas the average across the alternatives was just one-fifth that under single-user load.

Similarly, Figure 7 (right) compares the throughput of the four systems. Impala achieves from 8.7x up to 22x higher throughput than the other systems when 10 users submit queries from the interactive bucket.

7.4 Comparing against a commercial RDBMS

From the above comparisons it is clear that Impala is on the forefront among the SQL-on-Hadoop systems in terms of performance. But Impala is also suitable for deployment in traditional data warehousing setups. In Figure 8 we compare the performance of Impala against a popular commercial columnar analytic DBMS, referred to here as "DBMS-Y" due to a restrictive proprietary licensing agreement. We use a TPC-DS data set of scale factor 30,000 (30TB of raw data) and run queries from the workload presented in the previous paragraphs. We can see that Impala outperforms DBMS-Y by up to 4.5x, and by an average of 2x, with only three queries performing more slowly.

8. ROADMAP

In this paper we gave an overview of Cloudera Impala. Even though Impala has already had an impact on modern data management and is the performance leader among SQL-on-Hadoop systems, there is much left to be done. Our

[7] https://github.com/cloudera/impala-tpcds-kit

[8] There are several other SQL engines for Hadoop, for example Pivotal HAWQ and IBM BigInsights. Unfortunately, as far as we know, these systems take advantage of the DeWitt clause and we are legally prevented from presenting comparisons against them.

[9] http://blog.cloudera.com/blog/2014/05/new-sql-choices-in-the-apache-hadoop-ecosystem-why-impala-continues-to-lead/

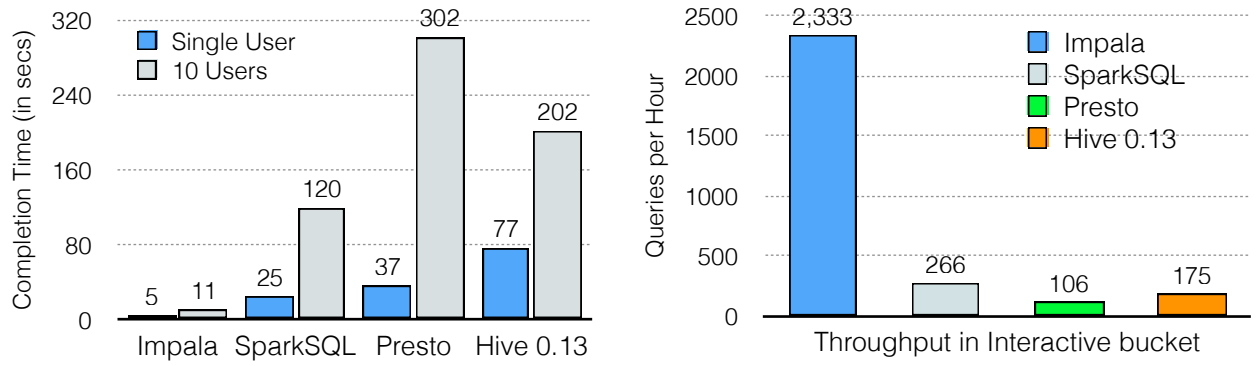

Figure 7: Comparison of query response times and throughput on multi-user runs.

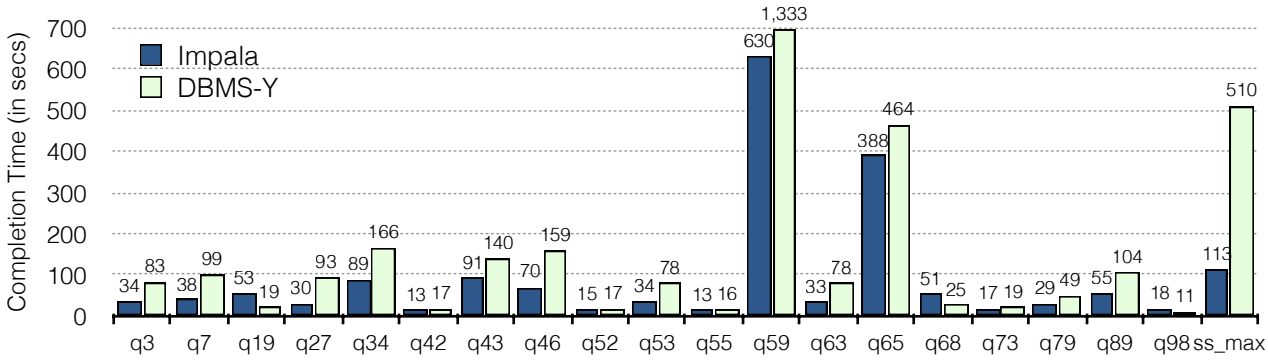

Figure 8: Comparison of the performance of Impala and a commercial analytic RDBMS.

roadmap items roughly fall into two categories: the addition of yet more traditional parallel DBMS technology, which is needed in order to address an ever increasing fraction of the existing data warehouse workloads, and solutions to problems that are somewhat unique to the Hadoop environment.

8.1 Additional SQL Support

Impala's support of SQL is fairly complete as of the 2.0 version, but some standard language features are still missing: set `MINUS` and `INTERSECT`; `ROLLUP` and `GROUPING SET`; dynamic partition pruning; DATE/TIME/DATETIME data types. We plan on adding those over the next releases.

Impala is currently restricted to flat relational schemas, and while this is often adequate for pre-existing data warehouse workloads, we see increased use of newer file formats that allow what are in essence nested relational schemas, with the addition of complex column types (structs, arrays, maps). Impala will be extended to handle those schemas in a manner that imposes no restrictions on the nesting levels or number of nested elements that can be addressed in a single query.

8.2 Additional Performance Enhancements

Planned performance enhancements include intra-node parallelization of joins, aggregation and sort as well as more pervasive use of runtime code generation for tasks such as data preparation for network transmission, materialization of query output, etc. We are also considering switching to a columnar canonical in-memory format for data that needs to be materialized during query processing, in order to take advantage of SIMD instructions [11, 13].

Another area of planned improvements is Impala's query optimizer. The plan space it explores is currently intentionally restricted for robustness/predictability in part due to the lack of sophisticated data statistics (e.g. histograms) and additional schema information (e.g. primary/foreign key constraints, nullability of columns) that would enable more accurate costing of plan alternatives. We plan on adding histograms to the table/partition metadata in the near-to-medium term to rectify some of those issues. Utilizing such additional metadata and incorporating complex plan rewrites in a robust fashion is a challenging ongoing task.

8.3 Metadata and Statistics Collection

The gathering of metadata and table statistics in a Hadoop environment is complicated by the fact that, unlike in an RDBMS, new data can show up simply by moving data files into a table's root directory. Currently, the user must issue a command to recompute statistics and update the physical metadata to include new data files, but this has turned out to be problematic: users often forget to issue that command or are confused when exactly it needs to be issued. The solution to that problem is to detect new data files automatically by running a background process which also updates the metadata and schedules queries which compute the incremental table statistics.

8.4 Automated Data Conversion

One of the more challenging aspects of allowing multiple data formats side-by-side is the conversion from one format into another. Data is typically added to the system in a struc-

tured row-oriented format, such as Json, Avro or XML, or as text. On the other hand, from the perspective of performance a column-oriented format such as Parquet is ideal. Letting the user manage the transition from one to the other is often a non-trivial task in a production environment: it essentially requires setting up a reliable data pipeline (recognition of new data files, coalescing them during the conversion process, etc.), which itself requires a considerable amount of engineering. We are planning on adding automation of the conversion process, such that the user can mark a table for auto-conversion; the conversion process itself is piggy-backed onto the background metadata and statistics gathering process, which additionally schedules conversion queries that run over the new data files.

8.5 Resource Management

Resource management in an open multi-tenancy environment, in which Impala shares cluster resource with other processing frameworks such as MapReduce, Spark, etc., is as yet an unsolved problem. The existing integration with YARN does not currently cover all use cases, and YARN's focus on having a single reservation registry with synchronous resource reservation makes it difficult to accommodate low-latency, high-throughput workloads. We are actively investigating new solutions to this problem.

8.6 Support for Remote Data Storage

Impala currently relies on collocation of storage and computation in order to achieve high performance. However, cloud data storage such as Amazon's S3 is becoming more popular. Also, legacy storage infrastructure based on SANs necessitates a separation of computation and storage. We are actively working on extending Impala to access Amazon S3 (slated for version 2.2) and SAN-based systems. Going beyond simply replacing local with remote storage, we are also planning on investigating automated caching strategies that allow for local processing without imposing an additional operational burden.

9. CONCLUSION

In this paper we presented Cloudera Impala, an open-source SQL engine that was designed to bring parallel DBMS technology to the Hadoop environment. Our performance results showed that despite Hadoop's origin as a batch processing environment, it is possible to build an analytic DBMS on top of it that performs just as well or better that current commercial solutions, but at the same time retains the flexibility and cost-effectiveness of Hadoop.

In its present state, Impala can already replace a traditional, monolithic analytic RDBMSs for many workloads. We predict that the gap to those systems with regards to SQL functionality will disappear over time and that Impala will be able to take on an every increasing fraction of pre-existing data warehouse workloads. However, we believe that the modular nature of the Hadoop environment, in which Impala draws on a number of standard components that are shared across the platform, confers some advantages that cannot be replicated in a traditional, monolithic RDBMS. In particular, the ability to mix file formats and processing frameworks means that a much broader spectrum of computational tasks can be handled by a single system without the need for data movement, which itself is typically one of the biggest impediments for an organization to do something useful with its data.

Data management in the Hadoop ecosystem is still lacking some of the functionality that has been developed for commercial RDBMSs over the past decades; despite that, we expect this gap to shrink rapidly, and that the advantages of an open modular environment will allow it to become the dominant data management architecture in the not-too-distant future.

References

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

[2] Apache. Centralized cache management in HDFS. Available at https://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html.

[3] Apache. HDFS short-circuit local reads. Available at http://hadoop.apache.org/docs/r2.5.1/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.

[4] Apache. Sentry. Available at http://sentry.incubator.apache.org/.

[5] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, 2007.

[6] A. Floratou, U. F. Minhas, and F. Ozcan. SQL-on-Hadoop: Full circle back to shared-nothing database architectures. *PVLDB*, 2014.

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

[8] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

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

[10] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, 2001.

[11] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU Acceleration: So much more than just a column store. *PVLDB*, 6, 2013.

[12] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SOCC*, 2013.

[13] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2, 2009.