A High-Performance Distributed Relational Database System for Scalable OLAP Processing

Jason Arnold
Illinois Institute of Technology
jarnold6@hawk.iit.edu

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

Ioan Raicu
Illinois Institute of Technology
iraicu@cs.iit.edu

***Abstract*—The scalability of systems such as Hive and Spark SQL that are built on top of big data platforms have enabled query processing over very large data sets. However, the per-node performance of these systems is typically low compared to traditional relational databases. Conversely, Massively Parallel Processing (MPP) databases do not scale as well as these systems. We present *HRDBMS*, a fully implemented distributed shared-nothing relational database developed with the goal of improving the scalability of OLAP queries. HRDBMS achieves high scalability through a principled combination of techniques from relational and big data systems with novel communication and work-distribution techniques. While we also support serializable transactions, the system has not been optimized for this use case. HRDBMS runs on a custom distributed and asynchronous execution engine that was built from the ground up to support highly parallelized operator implementations. Our experimental comparison with Hive, Spark SQL, and Greenplum confirms that HRDBMS's scalability is on par with Hive and Spark SQL (up to 96 nodes) while its per-node performance can compete with MPP databases like Greenplum.**

***Index Terms*—SQL, big data, distributed query processing**

I. Introduction

The increasing scale of data to be processed for analytics has brought traditional DBMS that scale only to a few nodes to their limits. Massively Parallel Processing (MPP) databases provide better scale out by parallelizing query processing across processors and nodes using a shared-nothing architecture. Systems like Greenplum [5], Teradata [3], and Netezza feature parallel execution engines that are specialized for processing relational queries. Recently, a new class of SQL engines has been built on top of Big Data [42] platforms such as MapReduce [18] and Spark [11], [49]. This class of systems include Hive [44], Spark SQL [47], Dremel [32], and many others. Big Data platforms utilize distributed file systems like HDFS for storage and resource managers like YARN [45] to schedule execution of tasks on a cluster. SQL engines built on top of Big Data platforms rely on the fault tolerance and load balancing techniques of these platforms for scalability. While these approaches provide better scale out (deployments on 1000s of nodes are not uncommon), their performance per-node is limited by the programming, storage, and execution model of the underlying platform. Both types of systems have in common that they optimize for OLAP, i.e., complex, mostly read-only, queries. Figure 2 compares systems in terms of scalability (scale-out) and per-node performance. Databases without intra-query parallelism (e.g., Postgres) do not scale to large clusters and also have limited per-node performance for complex queries, because they do not exploit the available parallelism on a single machine. MPP databases provide better per-node performance and better scalability. Big Data platforms provide scalability at the cost of per-node performance.

This work was supported in part by the National Science Foundation (NSF) under awards OCI-1054974 and OAC-1640864.

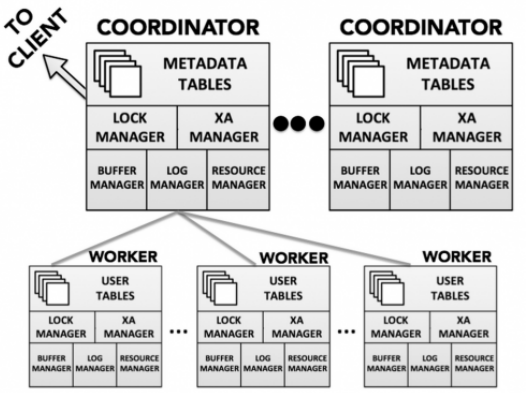


Fig. 1: Anatomy of a HRDBMS cluster

The goal of building HRDBMS (*highly-scalable relational DBMS*), the system we present in this work, is to build a DBMS with a per-node performance comparable to MPP databases as well as scalability comparable with Big Data platforms. Based on an analysis of scalability bottlenecks in MPP databases and Big Data platforms, we propose and implement novel techniques to address these bottlenecks. In the following we discuss the design of our system and the techniques we have developed for achieving scalability without sacrificing performance. An overview of our design decisions is shown in Figure 3. We indicate in parentheses after a feature whether it is typical for relational databases, typical for Big Data platforms, or is novel.

A. System Overview

Cluster Organization. As shown in Figure 1, an HRDBMS cluster consists of *coordinator* and *worker* nodes. Coordinators store metadata, handle communication with clients, and are responsible for query optimization, transaction coordination,

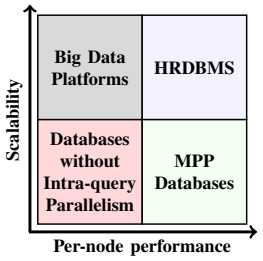


Fig. 2: Scalability vs. per-node performance trade-off

Feature	Implementation in HRDBMS
Storage	Page-oriented storage (**DB**) partitioned across nodes and disks; Memory caching using a parallel buffer manager (**DB**); Disk-resident index structures; Predicate-based data skipping (**N**); External Table Framework for accessing non-HRDBMS sources
Fault tolerance	Replicated storage is supported but not yet used for fault tolerance; Mid-query failures require query restart (**DB**); Temporary node failures are handled by ARIES-style log-based recovery (**DB**)
Resource Management	Coordinator nodes manage communication and distribution of work across workers; Worker nodes manage memory and degree of parallelism individually (**DB**).
Execution Engine	Custom distributed, pipelined (**DB**) execution engine with built-in support for relational operators; Inherent parallel implementation of all operator types; Extensive use of shuffle and other distributed implementations of operators (**BD**)
Optimization	A cost-based optimizer that applies heuristic and cost-based transformations (**DB**)
Communication	Enforcing a limit N_{max} on the number of neighbors a node has to communicate with for both hierarchical (**BD**) and n-to-m communication (**N**); Multiple coordinators for reliability and load-balancing client communication and transaction management

Fig. 3: Overview of HRDBMS design decisions. We indicate which features are typical for big data platforms (BD), typical for (distributed) relational databases (DB), or are novel (N).

and cluster-wide resource management. HRDBMS provides serializability using 2PL [9] and 2PC [35], a protocol that is implemented by many modern distributed databases such as Spanner [15]. Workers store data and execute queries. Systems such as Hadoop use a single node for handling client requests and storing metadata. These systems allow for additional stand-by masters which do not process requests. HRDBMS replicates metadata and statistics across all coordinators in a cluster, but allows multiple coordinators to process requests in parallel. Queries are executed across the workers using HRDBMS's custom execution platform. Query results are always routed to a client through the coordinator that planned the query.

Communication. A challenge faced by any distributed database system is that operations like shuffling require one node to communicate with a large number of nodes in the cluster. This is one of the primary scalability bottlenecks in MPP systems. The shuffle implementation of most Big Data platforms is designed to scale. However, scalability comes at the cost of performance since shuffle operations in these systems are blocking and most implementations persist data to local or distributed file systems. We develop new communication primitives that *enforce scalable communication topologies* for all communication in the system including query execution and transaction processing. Our approach enforces a hard limit N_{max} on the number of neighbors a node has to communicate with. Specifically, we propose two communication topologies - the *tree topology* for hierarchical communication (e.g., commit processing [35]) as well as a *n-to-m topology* for all-to-all communication (e.g., a shuffle). Our tree topology is similar to how systems like Dremel [32] implement hierarchical aggregation. To the best of our knowledge we are the first to generalize this idea to n-to-m communication. As a general rule, all operations involving communication in HRDBMS are non-blocking and do not spill to disk.

Storage and Locality. HRDBMS supports both row and columnar tables. Data is stored on the local disks of worker nodes using a page-oriented layout. Similar to MPP databases, tables are partitioned across nodes and disks based on a partitioning strategy that is selected at table creation time. Having full control over storage location enables our optimizer and execution engine to enforce data locality which is critical for performance and scalability. Page-oriented storage and support for disk resident index structures enables us to feasibly support DML operations and transactions. Data is cached in memory using a parallel buffer manager. We introduce *predicate-based data skipping* as a lightweight method for skipping pages during scan operations based on the results of previous scans. This method generalizes static schemes like small materialized aggregates [33], and could be retrofitted into existing database systems. We do not use a distributed file system such as HDFS for storage, because these file systems are typically append-only and their read/write throughput is less than native file operations. In addition to better read/write performance, we also gain full control over data locality by managing local storage directly. One advantage of Big Data platforms is that they can process queries over data from various sources without requiring the data to be ingested into the system. To combine the best of both worlds we have implemented an external table framework that allows HRDBMS to access a variety of external data sources, including HDFS.

Execution Engine. A major contributor to the poor performance of Hive and other Hadoop-based approaches is excessive materialization. Even systems such as Spark SQL still materialize data during shuffle operations per default. Similar to MPP systems we only materialize intermediate results to disk when necessary to save memory. Operator implementations in HRDBMS are inherently parallel. Similar to both Big Data platforms and MPP databases we spawn one scan thread for each fragment of a table, but the parallelism of other operators is controlled separately. HRDBMS features a cost-based query optimizer that uses statistics for cost estimation.

Fault Tolerance. Fault tolerance in HRDBMS is currently limited to Aries-style recovery [34] based on write-ahead logs maintained on each node. Metadata is replicated across all coordinators and kept in-sync using the 2PC protocol [35].

Resource management. Systems like YARN [45] and

Mesos [21] are frequently used to manage the resources in a cluster, e.g., to let multiple applications share the same cluster. We deliberately chose to let HRDBMS manage its own resources, because HRDBMS tasks dynamically resize their resource allocations based on their current requirements and availability of resources. Resource management in HRDBMS operates on three levels. At the cluster level, our query optimizer balances load and communication across workers. Workers monitor the resource usage on the machine they are running on and reduce the degree of parallelism for query operators if resources are scarce. Finally, operators can spill data to disk if necessary to limit memory consumption. This decentralization of resource management for lower level tasks is critical for scalability, because it avoids overloading coordinators with decisions that can be better made locally.

B. Contributions

Our key contributions are two-fold: (i) we demonstrate how a principled combination of traditional query processing techniques with ideas from distributed dataflow engines can improve performance and scalability leading to a system that is more than the sum of its components. However, to fully achieve our goal we had to (ii) develop novel techniques that overcome the remaining performance and scalability bottlenecks. We implement these ideas as a system called HRDBMS, a fully functional shared-nothing distributed DBMS. HRDBMS's open source codebase (`https://github.com/IITDBGroup/HRDBMS`) currently consists of about 170,000 lines of Java code.

Analysis of Bottlenecks and Principled System Design. We identify and address performance bottlenecks including non-scalable communication, blocking shuffle operations, and data locality not being exploited. By choosing and picking the most effective and scalable techniques from MPP databases and big data platforms, we can overcome many of these bottlenecks. This results in a new hybrid design which has both characteristics from distributed relational databases such as Greenplum as well as big data platforms such as Spark as discussed in Section I-A.

Novel Techniques. The principled combination of previous ideas however is not sufficient for achieving our goal. We additionally make the following technical contributions that improve the state-of-the-art in distributed query processing in several regards.

- **Scalable communication patterns for all operations:** Approaches like Dremel [32] reduce the number of neighbors a node has to communicate with for hierarchical operations such as aggregation. However, to the best of our knowledge, we are the first to enforce this for operations with n-to-m communication patterns such as a shuffle. For that we apply a variation of the binomial graph topology [4] to be able to enforce a constant limit on the number of neighbors a node has to directly communicate with. Since shuffle is a prevalent operation in distributed query processing (e.g., it is used to implement distributed versions of group-by aggregation and join), this translates into a significant improvement in scalability (e.g., see the discussion of our experimental results for TPC-H query 19 in Section VII).
- **Predicate-based Data Skipping:** Data skipping based on small materialized aggregates [33] stores for each page of a table the minimal and maximal values for each of the table's attributes. This information is used during query processing to skip a page if based on the minimal and maximal attributes values of records stored on this page, none of the records can fulfill the selection conditions of the query. We generalize this idea by caching at runtime which pages contain rows matching a query's predicate and then use this information to skip reading pages for future queries with the same or similar predicates. Since many workloads follow the 80-20 rule, i.e., 80% of the queries only access 20% of data, it is quite effective to cache query-specific information about relevant data.

Experimental Evaluation. We experimentally evaluate our system on a 96 node cluster comparing against Hive, Spark SQL, and Greenplum. Our results demonstrate that HRDBMS provides per-node performance competitive with Greenplum and exhibits scalability comparable with Hive and Spark SQL. Furthermore, HRDBMS performs well if data is significantly larger than the available main memory.

The remainder of this paper is organized as follows. We discuss related work in Section II. Section III discusses the system's storage engine and indexing capabilities. We introduce our distributed execution engine in Section IV. Section V covers HRDBMS's cost-based optimizer. Section VI covers concurrency control and recovery. We present experimental results in Section VII and conclude in Section VIII.

II. Related Work

HRDBMS builds upon the long history of relational query processing as well as on ideas from big data platforms. We did present a preliminary version of the system in [7], [8].

MPP Databases. MPP databases such Greenplum, Netezza, and Teradata distribute pipelined query processing across nodes using a shared-nothing architecture. MPP databases utilize local disks on worker nodes for page-oriented record storage and replication for fail-over. A single master node handles communication with clients and query optimization. In general these systems are high-performance, but are not designed to scale to large clusters. Several approaches pair MPP processing with HDFS storage (e.g., HAWQ [13]).

SQL on Big Data Platforms. *Hive* [44] is an SQL engine built on top of *Hadoop*. While inheriting the scalability of its host platform, the per-node performance of Hive is severely restricted by the excessive materialization of MapReduce and lack of support for exploiting data locality. Since MapReduce is largely I/O-bound, compressed file formats like *ORC* and *Parquet* can significantly improve performance [19]. While these techniques address some of the bottlenecks of MapReduce, others, such as limitations of the MapReduce shuffle

model, are inherent to the platform itself. Tez [39] improves MapReduce by allowing multiple mappers and reducers to be chained together without writing temporary data to HDFS and by reconfiguring dataflows at runtime. Impala [25] deploys daemons on HDFS data nodes which access the data stored locally on a node to ensure data locality and eliminate most of the overhead of HDFS. Spark SQL [6] is an SQL engine that is part of Spark. The system features an extensible optimizer. Even though most materialization is eliminated by using Spark, shuffle operations still write data to disk by default. Furthermore, the system has high memory requirements. Dremel [32] is designed for aggregation-heavy analytics across large clusters. The system performs well for aggregation queries with small result sets because of its serving-tree architecture. HRDBMS applies a generalization of this idea by dynamically organizing nodes in a suitable topology to compute aggregation, sort, and shuffle operations. Apache Flink [10], [20] is a streaming dataflow engine. Operators execute in parallel and slower operators can backpressure upstream operators to prevent unlimited growth of streams.

SQL on Key-value Stores. Recently, several SQL front-ends for NoSQL databases (key-value stores) such as Cassandra [26] and ZHT [29], [30] have been introduced. NoSQL databases are typically designed for high-throughput CRUD operations and cannot efficiently process batch operations [38] which is required for OLAP workloads. However, there are key-value stores such as BigTable [12] that are better suited as storage backends for relational engines. Spanner [15] is a database based on BigTable with support for strongly consistency. In Spanner, data is co-located within the key-value store backend based on a hierarchical nesting of tables. However, not all workloads can benefit from hierarchical co-location.

NewSQL. Another class of systems target scalable processing of OLTP workloads. These include Spanner [15], VoltDB [43] (which is a commercialization of the H-store [23] research prototype), FoundationsDB, Amazon Aurora, and many more. To scale-out, these systems horizontally partition tables across several nodes. Transactions that only access data stored on one node are handled locally while transactions involving multiple nodes use a distributed commit protocol such as 2PC [35] or a consensus protocol like Paxos [27] or Raft [37]. For example, Spanner [15] applies Paxos to ensure consistency among multiple replicas of a table fragment and 2PC to coordinate transactions that access more than one fragment. While transaction processing is not our main focus, HRDBMS nonetheless supports serializable transactions relying on a hierarchical version of the 2PC protocol. However, the performance of our current implementation is relatively poor.

III. Storage

HRDBMS stores data on the local filesystems of worker nodes. Tables may be hash- or range-partitioned, or duplicated across nodes. Within each node an additional level of partitioning is applied to spread data across the node's disks. The partitioning strategy for a table is selected by the user at table creation time. The coordinator nodes track how table data is distributed across workers and use this information to enforce that data is always read on the worker node(s) where it resides. This is in contrast to systems such as Hive or Spark SQL where locality can only be stated as a preference.

Block Storage. All index and table data in HRDBMS is stored in pages of configurable size (up to 64MB). Pages are compressed using LZ4 [14] (fast decompression).

Row and Column Storage. HRDBMS supports both row and columnar tables. The row-oriented table implementation is standard: each page contains a header with information about the rows stored on that page and their schema; the rest of the page contains data. Each row is assigned a physical row ID (RID) that consists of identifiers for the node, disk, and page on which the row resides plus a slot number identifying the row's position within the page. The user can specify that a table should be clustered on a list of attributes. Data is sorted during loading to enforce the clustering. DML operations do not respect the clustering. However, HRDBMS provides support for reorganizing tables to restore clustering.

Columnar tables are implemented as a row/columnar hybrid, similar to PAX [2]. All columns of a table are stored in a single file as a sequence of *page sets*. A page set for a table with n columns consists of n pages, each storing values of one column. Each page in a page set stores the same number of values which simplifies reconstruction of rows. A naïve implementation of the page-set approach would underutilize pages since the column with the largest values determines the number of rows that can be stored in a page set. We address this problem by 1) using Huffman encoding [22] of strings and LZ4 page compression and 2) using a Linux sparse file, i.e., free space on pages will (almost) not occupy any space on disk. The Linux sparse file approach allows us to directly access a certain page in a file without knowing the offsets of compressed pages within the file.

Buffer Manager. All access to data and index pages in HRDBMS is handled by a node's buffer manager. The buffer manager has the ability to dynamically grow and shrink the buffer pool as needed. The buffer pool of a node is partitioned into stripes - each managed by a stripe manager (separate thread). The assignment of pages to stripes is determined by a hash of the page number. The parallel nature of the buffer manager is hidden from its clients through a lightweight wrapper that forwards requests to the stripe managers. We use a variant of the standard clock algorithm for page eviction. Our implementation differs from the standard clock algorithm in that table scans periodically pre-declare the pages they will request in the near future and these pages will be prioritized by the clock algorithm. This approach is particularly effective if a large percentage of buffer operations stems from concurrent table scan operations as is common for OLAP workloads.

Predicate-based Data Skipping. During a table scan, HRDBMS keeps track of which pages do not contain any rows matching a predicate and records this information in a predicate cache associated with each individual page. This is maintained as a mapping $cache : P \rightarrow \{\theta_i\}$ from a page P to

a set of predicates θ_i. Subsequent table scan operations with a predicate θ can then skip each page P with $\theta \in cache(P)$. Furthermore, if a predicate θ logically implies one of the cached predicates for a page P, say predicate θ_i, then it is guaranteed that page P cannot contain any rows matching the predicate and, thus, can safely be skipped. Inserts in HRDBMS are *append-only* and updates are not executed in place. Thus, as long as a page or page set is full, any information we cache about it will be valid until the table is reorganized. Based on our experience with HRDBMS, a 10TB database with 1000 previously executed queries running on 10 nodes would contain about 250MB of predicate cache data per node. Predicate caches are periodically persisted to disk and loaded during database restarts. Many databases implement a concept known as min-max indexes where the database tracks the minimum and maximum value of a column on each page [33]. Our method is a generalization of min-max indexes.

Index Structures. HRDBMS supports B+-trees and disk-resident skip lists. Skip lists are fairly common in main-memory databases (e.g., [17]) because they can support non-blocking concurrent access. We map skip lists to disk as an append-only page file. Any new node inserted into a skip list is inserted to the current page and deletes are logical. In spite of its simplicity, this results in reasonable I/O performance for traversing the index if data is mostly inserted in batches.

External Data Sources. HRDBMS features an extensible external table framework which supports distributed access to external data sources. A *user-defined external table type* (*UET*) for a distributed data source can expose the horizontal partitioning of data to HRDBMS and the system will distribute scans of fragments of such a table across worker nodes. As a proof-of-concept we have implemented an external table type for reading CSV data from HDFS.

IV. Execution Engine

HRDBMS features a distributed dataflow engine with built-in support for relational operators. Our engine is inherently asynchronous and pipelines operations within and across nodes and processors. Per default, once data is read from disk, all operations execute in main memory, spilling to disk only where necessary. The engine enforces data locality - all table and intermediate data is accessed locally on the node where it resides. Importantly, HRDBMS's optimizer can place any operator on any worker node.

Pipelined Execution. Pipelining is a standard practice that avoids unnecessary materialization of intermediate results. Operators that need to consume their whole input before producing a result are called *blocking*. HRDBMS tries to avoid blocking operators, e.g., by choosing a non-blocking implementation of an operator or by splitting an operator into two phases where only one of the phases is blocking.

Enforcing Scalable Communication. A critical scalability bottleneck of many distributed SQL engines is that operations such as shuffle may require nodes to communicate with $\mathcal{O}(n)$ neighbors where n is the number of nodes in the cluster.

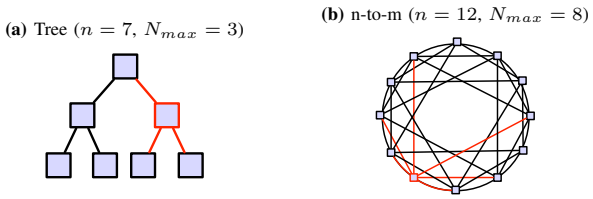


Fig. 4: Exemplary tree and n-to-m topologies

For large cluster sizes, the amount of resources occupied by opening and monitoring such a large number of sockets may already be a performance bottleneck. We support two strategies for enforcing a limit N_{max} on the number of network connections per node which we refer to as *tree-topology* and *n-to-m-topology*, respectively.

We distribute hierarchical computations using a tree topology with a maximal fan-out of $N_{max}-1$ (see Figure 4(a)). An example of a hierarchical operator is a distributed merge sort, where each merge phase can be processed by a separate level in such a tree. Note that this topology is similar to what has been referred to as a serve-tree in [32]. In this topology every node is only communicating with its parent and children. In addition to limiting the amount of network connections, the tree-topology also results in more evenly balanced load.

The second strategy is applied for operations that require n-to-m communication, e.g., a shuffle. To enforce the N_{max} limit, we use some nodes as intermediate communication hubs which forward data from senders to receivers. Our approach is based on the binomial graph topology [4]. In this topology, nodes are organized in a ring. Particularly, a node has links to nodes if the distance (on the ring) between the source and the destination is a power of 2, e.g., 0, 1, 2, 4, and so on. This topology has logarithmic diameter and node degrees. We determine the base of the powers based on the number of nodes n and N_{max}. The appropriate base b is computed as $b = n^{\frac{2}{N_{max}}}$. Figure 4(a) shows such a topology.

Non-blocking, Hierarchical Shuffle. A shuffle operation hash-partitions data across nodes. In Hadoop, reducers expect their input to be sorted by key. Hadoop's shuffle is a blocking operation because of this sorting step. Since many relational operator implementations such as a partition hash join can be performed on unsorted data, it is beneficial to make the sort optional. HRDBMS implements this idea as a non-blocking shuffle operation that avoids writing data to disk. A potential drawback of a non-blocking shuffle is the large number of concurrent network connections that are required - each node has a connection to each other node involved in the shuffle. We use our n-to-m topology to address this drawback and refer to the resulting operation as a *hierarchical shuffle*.

Operator Implementations. HRDBMS implements hash-based aggregation, optionally using a shuffle to assign a subset of the groups to each worker. Alternatively, each nodes pre-aggregates data available locally akin to a combiner in MapReduce. If pre-aggregation is used, then we use our tree topology where each node further aggregates the pre-aggregated results

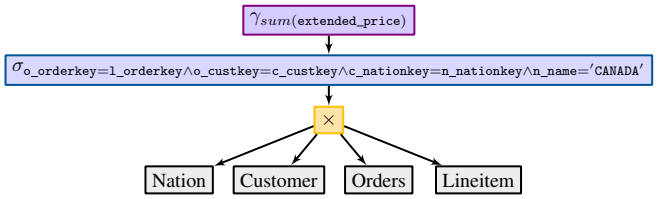


Fig. 5: Initial plan for the query from Example 1

received from its children. The same concept applies to sorting. We use a distributed implementation of an n-way merge sort based on the tree topology. Leaf nodes create sorted runs. Non-leaf nodes merge runs into larger runs. HRDBMS uses hash joins as long as at least one conjunct in the join condition is an equality comparison. Our hash join implementation uses bloom filters built over the join attributes of both inputs to filter rows to reduce communication (e.g., see [36]).

Operator Parallelism. HRDBMS uses both intra- and inter-operator parallelism to evaluate an execution plan on a worker node. The degree of parallelism for scan operators is determined based on the number of disks per worker. Complex operations such as joins and aggregations are executed in separate threads with a degree of parallelism that is adjusted independently. For example, an in-memory hash join has multiple threads reading records from its input, and each of these threads simultaneously probes the hash table.

Spilling to Disk. All operator implementations in HRDBMS have the ability to spill data to disk as needed. This decision is made at runtime based on available memory and statistics which predict the amount of memory that will be required.

V. Optimizer

HRDBMS has a cost-based optimizer that uses statistics for cost estimation. To reduce the size of the search space, we split optimization into multiple phases (e.g., join order is determined in the 1^{st} phase independent on how the query will be distributed across nodes) and make extensive use of heuristics. Optimization consists of the following phases:

- **1. Global Optimization Phase** - Perform heuristic and cost-based optimizations ignoring distribution.
- **2. Dataflow Conversion Phase** - Convert the query plan into a naïve distributed dataflow by ensuring data-locality of table and index scans. This is achieved by splitting scan operators into one scan per fragment of a table and placing the scan for each fragment on the node where the fragment resides. At this point all remaining operators of the query are placed on a coordinator node.
- **3. Dataflow Optimization Phase** - Optimize the dataflow by re-distributing operators from the coordinator to worker nodes and distributing the execution of individual operators across nodes.

In the following we describe each of these phases in detail using the example shown below.

Example 1. *Consider the following query over the TPC-H schema [16] that computes how much money Canadian customers have spent. Figure 5 shows the initial operator tree produced by HRDBMS's parser.*

```
SELECT sum(l_extendedprice)
FROM lineitem, orders, customer, nation
WHERE o_orderkey = l_orderkey
      AND o_custkey = c_custkey
      AND c_nationkey = n_nationkey
      AND n_name = 'CANADA'
```

1. Global Optimization Phase. HRDBMS takes advantage of decades of research and development in relational query optimization to build efficient execution plans. As a first step, conditional expressions are normalized and attribute equivalence classes are computed (e.g., equivalence classes are later used in join reordering). Afterwards, we apply standard heuristic transformations including pushing down selections [28] and projections. HRDBMS always de-correlates and un-nests nested subqueries if possible. Currently, we support the rewrites described by Kim [24]. We plan to implement more advanced techniques in the future (e.g., magic decorrelation as described by Seshadri et al. [41]).

Next we use statistics to apply cost-based join enumeration (determining the join order with the lowest cost) using a variant of the so-called greedy join enumeration algorithm [46] and choose which physical operators to use (e.g., table vs. index scan). Currently, our cost estimation module uses simple models for operator implementations. Parameters of these cost models are the estimated cardinalities of an operator's inputs (and for some operators also the estimated output cardinality). We estimate intermediate result sizes using standard techniques based on attribute-level statistics.

Afterwards, we apply transformations such as pushing group-by through joins. Our group-by transformation is inspired by Wong et al. [48]. Since this transformation is only sometimes beneficial, we make a cost-based decision on when to apply it. In the future, we will implement a wider variety of such transformations. Furthermore, we will investigate the use of non-exhaustive search strategies following an approach similar to cost-based transformations [1].

Example 2. *Reconsider the previous example query. Figure 6(a) shows a plan for this query that may be generated by the first phase of optimization. The selection on n_name has been pushed through the crossproduct and the crossproduct has been transformed into joins using conjuncts from the selection's condition. The system has decided to use a left-deep plan joining tables nation and customer first, then joining the result with the orders table and afterwards the lineitem table.*

2. Dataflow Conversion Phase. After a global plan has been produced by phase 1, the operator tree is converted into a naïve distributed dataflow by splitting each table scan operator into individual scan operators for the fragments of the table. The scan operator for a fragment is placed on the node storing the

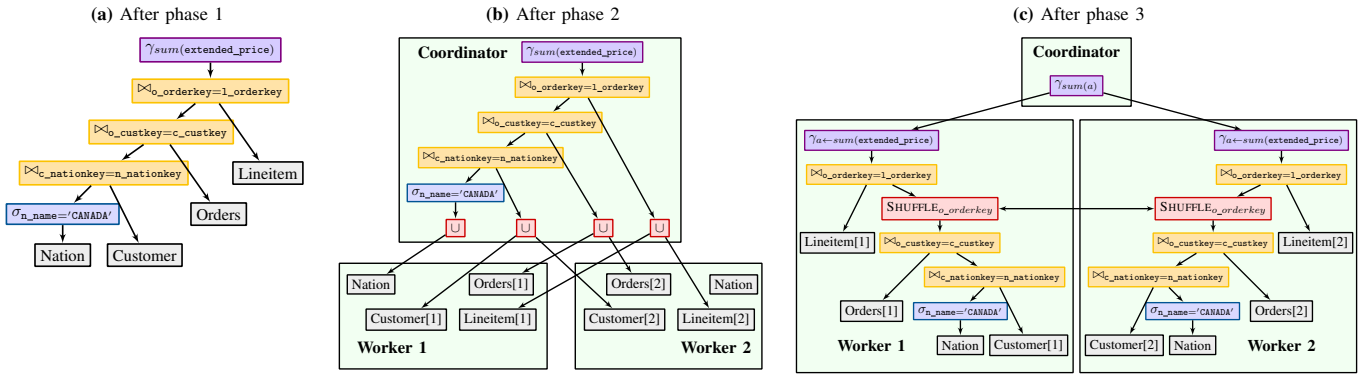


Fig. 6: Optimizing the query from Example 1. We show the query plan after each optimization phase.

fragment to enforce data locality. For tables that are replicated across the cluster, we assign the scan to a random worker. In this naïve dataflow translation all other operators are placed on the coordinator. Furthermore, union operators executed on the coordinator are used to merge the results of the fragment table scans on the worker nodes. The optimizer uses the selection conditions applied by the query to prune scans of fragments that are not needed to answer the query. For instance, if a table is range partitioned then we can exclude fragments for inequality and equality comparisons (e.g., $\mathsf{a} < 5$).

Example 3. *Continuing with Example 2, assume the query is executed on a cluster with one coordinator and two workers with one disk each. Table* nation *is replicated across both workers, table* customer *is hash-partitioned on* c_custkey*, table* orders *is hash-partitioned on* o_custkey*, and table* lineitem *is hash-partitioned on attribute* l_orderkey*. Figure 6(b) shows the dataflow graph produced by Phase 2 for the running example. We use $R[i]$ to denote the fragment of table R stored on node i. Note that locality is enforced for scans, but all remaining operators are placed on the coordinator.*

3. Dataflow Optimization Phase. In the dataflow optimization phase, we redistribute work from the coordinator to worker nodes by replacing operators with distributed implementations. We apply tree or n-to-m topologies to enforce the threshold N_{max} of neighbors a node has to communicate with. A major objective in this phase is to take further advantage of data locality and co-location - both existing locality based on partitioned table storage as well as data locality that is the result of shuffle operations. By striving to achieve locality we automatically reduce the amount of communication required and often reduce the number of shuffle operations.

Push Operators from Coordinator to Worker Nodes. We traverse the query plan produced by phase 2 bottom-up to distribute relational operators that are currently placed on the coordinator to the workers. Operators such as projection and selection can be pushed to the worker nodes directly. Equi-joins can be performed using a parallel distributed hash join as long as the data is partitioned across workers based on the join keys (or replicated across all workers corresponding to a broadcast join). At this stage we introduce shuffle operations to ensure co-location of join inputs. Likewise, sorting can be distributed by sorting independently at the workers followed by a merge phase on the coordinator. Instead of using a full sort, top-k queries (`LIMIT` with `ORDER BY`) are implemented using a local min-heap (assuming descending sort order) on each worker that computes the top-k rows on this worker. A newly arriving element is compared with the current minimum in the heap and only if the element is larger than the current minimun, then the minimum is removed and the new element is inserted into the heap. This guarantees that at any moment in time the heap stores the top-k elements we have seen so far. A max-heap is used at the coordinator (or another worker node) to merge the results of the individual top-k computations. There are corresponding rules for parallelization of all remaining relational operators. When multiple options for distributing an operator are available (e.g., aggregation) we make a greedy cost-based decision considering the cost of the complete plan for both options (at this stage we use a refined cost model that also incorporates communication cost). For instance, consider a query with an aggregation followed by a sort. Once we have determined which method to use for the aggregation we would never revert this decision.

Removing Unnecessary Shuffle Steps. Next, we remove unnecessary shuffle operations. For example, consider a query where one operation requires that data is hash-partitioned on a column a (all rows with $\mathsf{a} = x$ are on the same worker node) and another operation requires that data is partitioned on columns a (first) and column b (second). We can omit the second shuffle operation since guaranteeing that all tuples with the same a attribute value are present on the same worker automatically implies that all tuples with common a and b attribute values are present on the same worker.

Example 4. *Continuing with Example 3, the output of the final optimization phase is shown in Figure 6(c). The optimizer determines that the join between the nation and customer tables can be executed locally since the nation table is replicated across both nodes. Since the customer table is partitioned on* c_custkey *and the orders table is partitioned on* o_custkey*,*

the join between these two tables can be executed locally too. Finally, since lineitem is partitioned on l_orderkey, the result of the previous joins has to be partitioned on o_orderkey using a shuffle to evaluate the final join. Since the final aggregation does not use group-by, the optimizer decides to split the aggregation into a pre-aggregation at the worker nodes and a final aggregation on the coordinator.

VI. Concurrency Control

Unlike most databases built on-top Big Data platforms, HRDBMS supports DML operations and serializable transactions. HRDBMS runs a *lock manager*, *transaction manager*, and *log manager* on every coordinator and worker node. Additionally, each coordinator runs an *XA manager* that is responsible for coordinating commits using a hierarchical two-phase commit (2PC) protocol. While HRDBMS does support serializable transactions and DML, this has not been our main focus, and thus the performance and scalability of OLTP has not been thoroughly evaluated yet.

Lock Manager. HRDBMS achieves serializability and weaker isolation levels (such as repeatable read and read committed) via page-level locks using two lock-modes (shared and exclusive). We use a standard SS2PL locking protocol [9] (i.e., locks are held until transactions commit) to ensure serializability. The lock manager running on a node is only responsible for granting locks on that node. If a lock request fails on a node (due to deadlock or timeout), that lock manager will inform the XA manager on the coordinator handling the transaction. The XA manager will then initiate a cluster-wide rollback of the transaction. The lock managers currently implement local deadlock detection via a *wait-for graph* [31] and timeout to prevent deadlocks with multiple nodes involved. By default deadlock detection runs once a minute.

XA Manager and Transaction Manager. The XA manager on a coordinator acts as the global transaction manager keeping track of all active transactions that initiated from that coordinator. For each transaction it tracks which nodes are involved in this transaction. If a transaction T is to be committed or rolled back, the XA manager initiates a two-phase commit (2PC) process. HRDBMS uses a hierarchical 2PC protocol similar to the one used in System R* [35], i.e., all 2PC communication is based on our tree topology. The advantage of our tree topology is that messages can be quickly broadcasted to a large number of nodes, and that responses are aggregated before they are returned to the coordinator. This effectively reduces the amount of work and network communication at the coordinator. First, a `PREPARE` message is sent to all nodes involved in the transaction. Each node prepares the commit (or signals abort) and waits for a response from all nodes to which it forwarded the prepare message (its children in the tree topology). If the prepare processing on a node is successful and it receives positive responses from all of its children, then it returns a positive response to its parent. Only if all responses are positive, the coordinator sends a `COMMIT` message to all involved nodes. When a node receives a `PREPARE`, `COMMIT`, or `ROLLBACK` message, the transaction manager on that node that processes the necessary actions. This includes 1) requesting the buffer manager to unpin pages, 2) requesting the lock manager to release locks, and 3) requesting the log manager to persist entries to the write-ahead log.

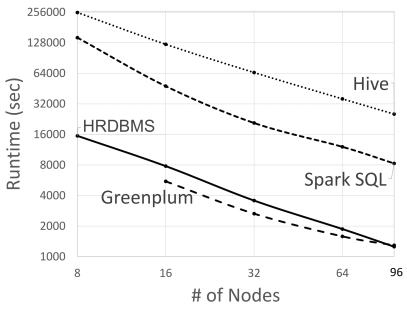
Log Manager. Each node has a log manager that manages the *write-ahead log* (*WAL*) on that node. In the case of worker nodes, the WAL is tracking changes to user data. For the coordinator nodes, the WAL is tracking changes to the system metadata tables. The log managers on the coordinator nodes also manage an additional log called the *XA log*. This log stores `PREPARE`, `COMMIT`, and `ROLLBACK` records as required by the 2PC protocol. The XA log is used to deal with worker node failures. When a worker node restarts, it attempts recovery from its own WAL. In some cases this log may not have enough information to determine the global state of a transaction. For example, if the last message that a worker received before it crashed was a `PREPARE` message for a transaction T, then it does not know if the consensus was to commit or rollback transaction T. The `PREPARE` record stored in the worker's WAL stores which coordinator is responsible. The worker contacts this coordinator to determine whether it should commit or rollback the transaction. While HRDBMS does not yet implement recovery for permanent node failures, it can recover the database state after a failed node (worker or coordinator) is brought back online.

Synchronization of Coordinator Metadata. In a deployment of HRDBMS with multiple coordinator nodes, the system metadata tables must be kept in sync across the coordinators. To achieve this, all insert, update, and delete statements that access metadata tables have to finish successfully on all coordinators in order to be considered successful. This means that a transaction changing metadata is not allowed to commit unless its DML operations have completed successfully on every coordinator. This is achieved using the 2PC protocol described above where the coordinator receiving a DDL statement coordinates the protocol.

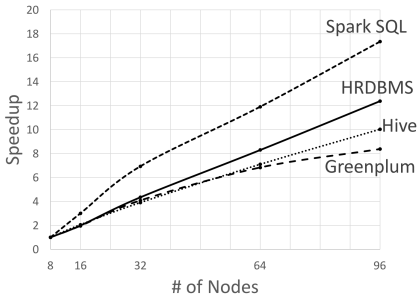
VII. Experiments

Setup & Workloads. We ran the TPC-H benchmark [16] query workload over a 1TB instance (SF1000) on setups ranging from 8 to 96 nodes. All tests were run on the Cooley cluster at Argonne National Laboratory. Each node has two Intel Haswell E5-2620 v3 processors, for a total of 12 cores and 384GB RAM. The nodes are connected via FDR Infiniband. All data, including temporary data, was placed on the GPFS filesystem [40] that these nodes share. All of the databases that we tested are designed to operate with local filesystems. Since the system we had access to only has a shared filesystem, we had to make sure that the shared filesystem was not a bottleneck in our experiments. We did extensive testing to determine appropriate sizing, i.e., as how many local disks should the GPFS filesystem be represented as on each node. We determined that we could allow each node to mimic 2

(a) total runtime



(b) Speed-up (relative to 8 nodes)



(c) Step-wise Speed-up

System	Efficiency Relative to Linear 8 to 16	16 to 32	32 to 64	64 to 96	8 to 96
Hive	103%	95%	91%	94%	83%
Spark	150%	115%	86%	97%	145%
Greenplum		104%	84%	82%	70%
HRDBMS	99%	109%	95%	99%	103%

System	Speed-up 8 to 16	16 to 32	32 to 64	64 to 96	8 to 96
Hive	2.06	1.90	1.82	1.41	10.02
Spark	3.01	2.31	1.72	1.46	17.34
Greenplum		2.08	1.67	1.23	8.37
HRDBMS	1.99	2.18	1.91	1.49	12.37

Fig. 7: 1TB TPC-H experiments: (a) total runtime, (b) speed-up relative to 8 nodes, and (c) step-wise speed-up

(a) 8 nodes

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22	Σ w/o Q9+Q18	Σ	Speed-up
HRDBMS	**844**	227	472	481	738	**85**	**471**	854	**2,186**	445	122	**226**	**20**	**50**	101	2,262	**2,580**	872	**308**	1,889	263	10,730	**15,496**	**16.41 x**
Hive	5,141	2,010	9,766	7,367	14,337	1,788	21,625	12,838	45,525	7,181	443	4,904	3,942	4,546	3,584	29,331	27,952	11,492	9,812	29,108	1,649	180,864	254,341	1.00 x
Spark SQL	3,182	497	7,410	3,107	9,357	1,826	8,675	9,107	13,310	4,038	791	2,923	2,067	1,886	836	17,911	18,845	6,186	3,452	27,718	640	111,609	143,764	1.77 x
Greenplum	868	**104**	**372**	**296**	**384**	216	569	**489**	OOM	**304**	**75**	312	255	497	**91**	**1,508**	OOM	**343**	410	**1,086**	**115**	**8,294**	N/A	N/A

(b) 96 nodes

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22	Σ	Speed-up
HRDMBS	83	27	41	46	56	**8**	**40**	65	121	33	40	**20**	**2**	**7**	**11**	145	**194**	77	**25**	151	61	**1,253**	**20.26 x**
Hive	789	275	1,044	911	1,445	156	2,176	1,337	4,215	944	179	797	622	678	510	1,744	2,453	1,068	986	2,588	468	25,385	1.00 x
Spark SQL	308	144	292	270	385	191	655	429	898	316	184	279	202	210	170	606	475	308	676	1170	121	8,289	3.06 x
Greenplum	**75**	**16**	**32**	**26**	**35**	19	43	**47**	**88**	**32**	**12**	29	24	46	11	**105**	433	**31**	90	**85**	**14**	1,293	19.63 x

Fig. 8: 1TB TPC-H per query runtime, total runtime (sec), and speed-up over slowest system at 8 vs. 96 nodes with lowest runtimes in bold. Since Greenplum did run out of memory for two queries on 8 nodes, we also show totals w/o these queries.

local disks for data storage and 2 local disks for temporary data using the GPFS filesystem, and that this could be scaled to 96 nodes in the Cooley environment without becoming a bottleneck. Since in our setup, data is distributed quite evenly across nodes, I/O operations also were quite evenly distributed. That is, the use of GPFS was not hiding load balancing issues. The main purpose of our evaluation is to compare the scalability and per-node performance of our system over a typical OLAP workload against main memory and disk-resident SQL engines built on top of Big Data platforms as well as against an MPP database system. We chose to test with Hive V1.2.1, Spark SQL V1.6.0, and Greenplum V4.3.99. Since we also wanted to evaluate system performance when the size of data exceeds available main memory, we restricted each node to use 24GB of RAM.

TPC-H. For our evaluation we use the TPC-H benchmark at the 1TB scale. The benchmark consists of 22 complex decision support queries. We ran 21 of the 22 queries, skipping the one that involved an outer join as outer joins have not been implemented in HRDBMS yet.

As mentioned before, we ran the benchmark on Hive, Spark SQL, Greenplum, and HRDBMS. Figure 7a shows the total elapsed time (wall clock time) for all TPC-H queries for each system when scaling from 8 to 96 nodes (log-log scale). Note that no result is shown for Greenplum at 8 nodes because the system ran out of memory for some of the queries. Figure 7b shows the speedup for each system relative to performance at 8 nodes. For computing the speedup, we used the elapsed time of the 19 queries that did finish at 8 nodes on Greenplum. Figure 7c shows stepwise speed-up for all systems (bottom) and stepwise speed-up as a percentage of linear speed-up on top (e.g, doubling the number of nodes decreases the runtime by 50%). Figure 8 shows individual query runtimes for 8 (top) and 96 nodes (bottom).

In general, Spark SQL is several times faster than Hive in our evaluation, HRDBMS is several times faster than Spark SQL, and Greenplum is 15% - 30% faster than HRDBMS. The performance gains over Spark are due to our non-blocking shuffle without materialization, scalable n-to-m topology for shuffles, predicate-based data skipping, and enforced data locality which allows us to often avoid shuffles for joins (and other operations). At 8 nodes, the performance for Spark SQL is much worse compared to the other cluster sizes. This is due to the fact that Spark uses excessive amounts of memory which results in high JVM garbage collection overhead. The poor performance of Spark SQL at 8 nodes artificially inflates its speedup results. Thus, we also show step-wise speed-up in Figure 7(c). While Spark SQL exhibits decent speedup, HRDBMS did even better. At the other end of the cluster size spectrum, we observe that Greenplum runs into scalability issues at 64 nodes and had significant problems scaling to 96 nodes. The other databases, HRDBMS included, scaled well to 96 nodes. In fact at 96 nodes, HRDBMS outperformed Greenplum by 3%.

Per Query Comparison with Greenplum. Since Greenplum was most competitive in terms of performance, we take a closer look at the performance of individual queries comparing against this system. Query 1 exhibits similar runtime on

# Nodes	Greenplum (runtime)	(speed-up)	HRDBMS (runtime)	(speed-up)
8	OOM		2,580	
16	856	1.00	1,499	1.00
32	395	2.17	731	2.05
64	351	2.44	319	4.70
96	433	1.98	194	7.73

Fig. 9: TPC-H query Q18 runtime (sec) and speed-up relative to 16 nodes (in parentheses) for Greenplum and HRDBMS.

HRDBMS and Greenplum at both 8 nodes and 96 nodes. This indicates that both systems have similar table scan performance and aggregation performance. Note that we use columnar tables for both systems. HRDBMS performs much better than Greenplum for queries 6, 14, 15, and 20. This is due to predicate-based data skipping. Greenplum outperforms HRDBMS on queries 2, 11, and 21 which involve correlated subqueries. Both Greenplum and HRDBMS unnest and decorrelate these subqueries, however there is an opportunity in these queries for reuse of intermediate results which HRDBMS does not currently exploit. Greenplum outperforms HRDBMS on query 19. Query 19 involves a complex predicate that becomes massive when converted into conjunctive normal form. Greenplum reorders the conjunctions after the conversion such that tuples are eliminated as early as possible. HRDBMS currently does not apply this optimization. Query 22 is an aggregation over a single table. The query contains two nested subqueries - one is a correlated `NOT EXISTS` while the other one is an uncorrelated scalar subquery (it returns a single value). Greenplum outperforms HRDBMS on this query, because it caches the result of the scalar subquery while, as mentioned above, HRDBMS does not yet support caching of intermediate results.

Figure 9 shows the runtime and speed-up for query 18. Up to 32 nodes, Greenplum outperforms HRDBMS. For higher number of nodes HRDBMS outperforms Greenplum. At 96 nodes, HRDBMS significantly outperforms Greenplum on this query. This query involves (semi-)joins producing large intermediate results, and an aggregation with 1.5 billion groups (for the 1TB instance). HRDBMS benefits from our n-to-m topology and from implementing the group-by using a shuffle.

3TB TPC-H Instance. In this experiment, we increased the size of the dataset to 3TB on 8 nodes. For this configuration the data is about 15 times larger than memory prior to compression. Given that Greenplum failed with out-of-memory errors for two queries at the 1TB scale at 8 nodes, it is not surprising that the same queries (Q9 and Q18) also failed at 3TB. In addition, Spark SQL failed with out-of-memory errors for the same queries at 3TB. We assume that Hive would have been able to complete all of the queries successfully, but based on our results for the 1TB TPC-H instance, we estimate that runtime of the entire benchmark would have taken close to 9 days to complete for the 3TB instance. HRDBMS, on the other hand, completed all 21 queries successfully taking 2.85x longer than the runtime for 1TB (for a total of $\sim$12 hours).

Current System Versions. To test how HRDBMS compares to newer versions of the tested systems, we repeated the 8 node experiments utilizing the full memory of the nodes (384GB) using Hive 2.1.0 (Hadoop 2.7.3) on Tez 0.8.4, Spark 2.0.1 (Hadoop 2.7.3), and Greenplum (V4.3.99). The total runtimes for each system are shown below. Spark SQL runtime is improved by $\sim$40%, HRDBMS by $\sim$12%, and all queries finished successfully with Greenplum. We observed a 3.7x speedup for Hive on Tez compared to Hive. However, HRDBMS still outperforms Hive on Tez by a factor of 2.9.

	Hive on Tez	Spark SQL	Greenplum	HRDBMS
Runtime (sec)	39,228	86,227	**10,186**	13,621

Summary. These results demonstrate that we were able to meet our main goal in developing HRDBMS. We have developed a distributed query execution engine capable of per-node performance on par with a traditional MPP relational database and with scalability on par with systems such as Hive and Spark SQL up to scales of 96 nodes.

VIII. Conclusions and Future Work

We present HRDBMS, a shared-nothing distributed database aimed to achieve good per-node performance as well as scalability. HRDBMS achieves this goal though a careful combination of traditional query processing techniques with techniques typically found in scalable dataflow systems. Achieving our goal required innovations including enforcing scalable communication patterns, predicate-based data skipping, and aggressive parallelization of all parts of the engine. In future work, we plan to implement fault tolerance to be able to recover from the complete loss of a node or disk and support iterative dataflows for machine learning and complex analytics.

References

[1] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. Cost-based query transformation in Oracle. In *VLDB*, pages 1026–1036, 2006.

[2] Anastassia Ailamaki, David J DeWitt, and Mark D Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB Journal*, 11(3):198–215, 2002.

[3] Mohammed Al-Kateb, Paul Sinclair, Grace Au, and Carrie Ballinger. Hybrid row-column partitioning in Teradata®. *PVLDB*, 9(13):1353–1364, 2016.

[4] Thara Angskun, George Bosilca, and Jack Dongarra. Binomial graph: A scalable and fault-tolerant logical network topology. In *ISPA*, pages 471–482, 2007.

[5] Lyublena Antova, Amr El-Helw, Mohamed A Soliman, Zhongxian Gu, Michalis Petropoulos, and Florian Waas. Optimizing queries over partitioned tables in MPP systems. In *SIGMOD*, pages 373–384, 2014.

[6] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[7] Jason Arnold, Boris Glavic, and Ioan Raicu. HRDBMS: A NewSQL database for analytics. In *CLUSTER*, pages 519–520, 2015.

[8] Jason Arnold, Boris Glavic, and Ioan Raicu. HRDBMS: Combining the best of modern and traditional relational databases. *CoRR*, abs/1901.08666, 2018.

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Data Eng. Bull.*, page 28, 2015.

[11] Bill Chambers and Matei Zaharia. Spark: The definitive guide big data processing made simple. 2018.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *TOC*, 26(2):4:1–4:26, June 2008.

[13] Lei Chang, Zhanwei Wang, Tao Ma, Lirong Jian, Lili Ma, Alon Goldshuv, Luke Lonergan, Jeffrey Cohen, Caleb Welton, Gavin Sherry, et al. HAWQ: a massively parallel processing SQL engine in Hadoop. In *SIGMOD*, pages 1223–1234, 2014.

[14] Yann Collet. LZ4: Extremely fast compression algorithm. `code.google.com`. Accessed: 2013, Jan 1.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, et al. Spanner: Google's globally-distributed database. In *OSDI*, pages 261–264, 2012.

[16] Transaction Processing Performance Council. TPC-H benchmark specification. `http://www.tcp.org/hspec.html`, 2008.

[17] Tyler Crain, Vincent Gramoli, and Michel Raynal. No hot spot non-blocking skip list. In *ICDCS*, pages 196–205, 2013.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[19] Avrilia Floratou, Umar Farooq Minhas, and Fatma Ozcan. SQL-on-Hadoop: Full circle back to shared-nothing database architectures. *PVLDB*, 7(12), 2014.

[20] Diego García-Gil, Sergio Ramírez-Gallego, Salvador García, and Francisco Herrera. A comparison on scalability for batch big data processing on Apache Spark and Apache Flink. *Big Data Analytics*, 2(1):1, 2017.

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[22] David A Huffman et al. A method for the construction of minimum-redundancy codes. *PIRE*, 40(9):1098–1101, 1952.

[23] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[24] Won Kim. On optimizing an SQL-like nested query. *TODS*, 7(3):443–469, 1982.

[25] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[26] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS*, 44(2):35–40, 2010.

[27] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[28] Alon Y Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *VLDB*, pages 96–107, 1994.

[29] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *IPDPS*, 2013.

[30] Tonglin Li, Xiaobing Zhou, Ke Wang, Dongfang Zhao, Iman Sadooghi, Zhao Zhang, and Ioan Raicu. A convergence of key-value storage systems from clouds to supercomputers. *Concurrency and Computation: Practice and Experience*, 28(1):44–69, 2015.

[31] Philip P Macri. Deadlock detection and resolution in a CODASYL based data management system. In *SIGMOD*, pages 45–49, 1976.

[32] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[33] Guido Moerkotte. Small Materialized Aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487, 1998.

[34] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1):94–162, 1992.

[35] C Mohan, Bruce Lindsay, and Ron Obermarck. Transaction management in the R* distributed database management system. *TODS*, 11(4):378–396, 1986.

[36] James K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, 1990.

[37] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *USENIX ATC*, pages 305–319. USENIX Association, 2014.

[38] Julian Rith, Philipp S Lehmayr, and Klaus Meyer-Wegener. Speaking in tongues: SQL access to NoSQL systems. In *SAC*, pages 855–857, 2014.

[39] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache Tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*, pages 1357–1369, 2015.

[40] Frank B Schmuck and Roger L Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, volume 2, pages 231–244, 2002.

[41] P. Seshadri, H. Pirahesh, and T.Y.C. Leung. Complex Query Decorrelation. In *ICDE*, pages 450–458, 1996.

[42] Ion Stoica and et al. The berkeley data analysis system (BDAS): An open source platform for big data analytics. Technical report, University of California, Berkeley, 2017.

[43] Michael Stonebraker and Ariel Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[44] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A warehousing solution over a Map-Reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[45] Vinod Kumar Vavilapalli and et al. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, page 5, 2013.

[46] Eugene Wong and Karel Youssefi. Decomposition - a strategy for query processing. *TODS*, 1(3):223–241, 1976.

[47] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[48] Weipeng P Yan, Per-Bike Larson, et al. Eager aggregation and lazy aggregation. In *VLDB*, volume 95, pages 345–357, 1995.

[49] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.