DCODE: A Distributed Column-Oriented Database Engine for Big Data Analytics

Yanchen Liu[1], Fang Cao[1(✉)], Masood Mortazavi[1], Mengmeng Chen[1], Ning Yan[1], Chi Ku[1], Aniket Adnaik[1], Stephen Morgan[1], Guangyu Shi[1], Yuhu Wang[2], and Fan Fang[2]

[1] Huawei Innovation Center, Santa Clara, CA, USA
{Yanchen.Liu,Fang.Cao,Masood.Mortazavi,Mengmeng.Chen,Yan.NingYan,
Chi.Ku,Aniket.Adnaik,Steve.Morgan,shiguangyu}@huawei.com
[2] Huawei Incubation Center, Hangzhou, China
{wangyuhu,fangfan}@huawei.com

Abstract. We propose a novel Distributed Column-Oriented Database Engine (DCODE) for efficient analytic query processing that combines advantages of both column storage and parallel processing. In DCODE, we enhance an existing open-source columnar database engine by adding the capability for handling queries over a cluster. Specifically, we studied parallel query execution and optimization techniques such as horizontal partitioning, exchange operator allocation, query operator scheduling, operator push-down, and materialization strategies, etc. The experiments over the TPC-H dataset verified the effectiveness of our system.

Keywords: Distributed DBMS · Data processing · Query optimization

1 Introduction

With data collection methods continuously evolving, the demand for analytic results from the data we collect also increases. In scientific research, decision support, business intelligence, medical diagnosis, financial analysis, and numerous other fields it has become more crucial than ever to gain insights from organized data.

The sheer size of data in modern analytic tasks on large datasets makes using traditional row-based database systems quite impractical. On the other hand, column-based engines have shown better performance results for analytic queries [10,17] primarily due to two reasons:

1. Only the specific columns needed have to be retrieved, which significantly reduces the amount of I/O dead weight when compared with the traditional row-based query processing model.
2. More sophisticated optimizations such as cache-conscious processing, vector compression and vectorized execution can be used, resulting in better use of modern CPUs' capabilities.

I. Khalil et al. (Eds.): ICT-EurAsia 2015 and CONFENIS 2015, LNCS 9357, pp. 289–299, 2015.
DOI: 10.1007/978-3-319-24315-3_30

As such, column storage techniques have been used not only in relational database systems (broadly supporting SQL) but also in other large-scale data processing systems, such as Google Dremel [16] and Apache Drill [1].

Many existing systems for large-scale data processing have been built using Hadoop. Hive [2] and YSmart [14] translate analytic queries into MapReduce jobs to be processed using Hadoop; others such as Cloudera Impala [3] and Facebook Presto [4] generate query equivalent operations directly over HDFS storage and bypass the MapReduce framework to improve query efficiency. While these systems achieve high scalability and reliability from their underlying Hadoop components, they nevertheless face the translation overhead from analytic queries to either MapReduce jobs or HDFS operations and miss many optimization opportunities frequently used by modern database systems. Moreover, these systems exhibit significant limitations on the types of SQL queries supported. Another system approaching scalable query processing is HadoopDB [9], which builds a hybrid engine taking advantage of the scalability of Hadoop as well as the full-fledged query processing capability of a DBMS but at the cost of maintaining two systems at the same time.

Motivated by pioneering work on database parallelism, we developed our analytic query engine **DCODE** on the following foundation: parallel processing of columnar data. Our original objective was to develop an efficient parallel processing engine for analytic workloads. Along the way, we investigated novel techniques, all implemented and verified in experiments. The major challenges and our responses can be outlined as follows: (i) Given column data distributed across multiple servers, the query processing engine has to be aware of the partitioning scheme so that it can dispatch tasks in a query plan to the proper servers for parallel processing. (ii) If a query contains a join operator over columns that differ in their partitioning keys or a group-by operator over non-partitioning keys, it is impossible to complete the execution without re-distributing or re-shuffling the original data. We address this issue by allocating a pair of exchange operators in the distributed query plan whenever re-shuffling is a must. Furthermore, as the dispatched query plans are running in parallel asynchronously, a light-weight scheduler is employed so that data dependencies between distributed operators are always guaranteed. (iii) Materialization strategy in a distributed column database engine has gone beyond the meaning of a non-distributed version [8]. Here we studied not only when the column data needs to be stitched together into rows, but also when a particular column requiring a re-shuffling operation has to have its value vectors re-shuffled during query execution. (iv) Finally, we investigated optimization opportunities such as remote query compilation, operator push-down, and optimal table selection for partitioned, multithreaded processing, which affect processing efficiency significantly in many types of analytic queries based on our complementary empirical investigations.

The rest of this paper is organized as follows. Section 2 describes major technique components and optimizations in the DCODE system. Section 3 briefly presents implementation details and evaluation results of the DCODE system. Our conclusions are presented in Sect. 4.

2 System Overview

2.1 Computing Architecture

The DCODE system consists of multiple shared-nothing commodity server nodes interconnected by high bandwidth Ethernet as shown in Fig. 1. Each server node is running an independent, full-fledged engine instance, so that users can submit queries to any node in the system for distributed processing. The server node accepting user query will be in charge of coordinating all servers involved in query processing progress. After query processing is completed, it will also be in charge of aggregating partial results from all servers and presenting final results to users.

Each server node employs a query parser module which is capable of parsing query text into a query plan where each operator is an atomic sequence of low-level operations over columnar data. A scheduler module decomposes the query plan into multiple sub-plan tasks and assigns those tasks to corresponding servers that possess the column data. It also arranges the execution order of all tasks to ensure that data dependencies are always satisfied across multiple tasks. An access engine executes the operations over the column storage and generates results. Many optimization techniques are used in an optimizer for further improving DCODE's processing efficiency.

2.2 Parallel Query Execution

Horizontal Partitioning. The DCODE system is capable of analytic query processing over a distributed column storage. For the purpose of parallel query execution, it partitions large tables across multiple server nodes horizontally according to either a range-based or a hash-based partition key values from the *partitioning* column or attribute. All other columns of the same table are co-partitioned horizontally with the same partitioning column. Small tables whose sizes are below a certain threshold are simply duplicated across all server nodes for saving the cost of potential data transmission over network.

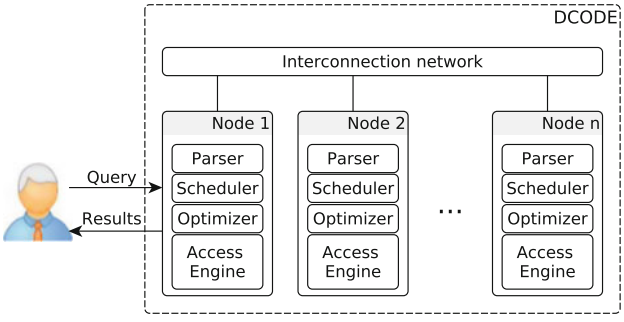

Fig. 1. Architecture of DCODE

The partitioning column can be selected either manually or in a principled manner based on a query workload model to minimize data transmission cost. For example, choosing two columns of different tables that appear in an equi-join query condition as partitioning columns will increase data affinity and eliminate re-shuffling transmission cost. However, an optimal horizontal partitioning scheme of a database containing multiple tables will vary for different query workloads and data distributions. Thus, it is hard to achieve [18].

Exchange Operator Allocation. One key issue towards parallel query execution in DCODE is to deal with incompatible keys. In a distributed query plan, the input values of an operator might be partitioned in a different way than needed by the operator. For example, when there is a join operator over two columns differing in their partition keys, no matter whether these keys are from the partitioning columns or not, they cannot be joined directly. Instead, the values from one joining key have to be re-distributed using the same manner as the other, such as re-hashing based on a hash function. A similar situation happens when we have a group-by operator over a non-partitioning column. We abstract this re-distributing or re-shuffling process in an *exchange* operator in a distributed query plan, adopting notion similar to the Volcano model [12] for parallel query processing.

In principle, a pair of exchange operators, i.e., `exchange_p` (producer) and `exchange_c` (consumer) will be allocated in a distributed query plan where data re-shuffling is a must. Operator `exchange_p` obtains intermediate results from the previous operator and stores them in a local memory pool, which are then re-shuffled and consumed by a remote `exchange_c` operator based on a new partitioning criterion required by the next operator. Note that we employ two exchange operators instead of one due to the fact that a consumer operator might absorb output results from multiple remote producer operators and they might be running in different operating system processes on different server nodes. An example of allocating exchange operators is shown in Fig. 2, where the original plan is to join two tables `T1` and `T2` on keys `T1.a` and `T2.b` (in Fig. 2a). Assuming these two keys are incompatibly partitioned, they cannot be locally joined. A pair of exchange operators are allocated in a distributed query plan for data re-shuffling (shaded in Fig. 2b) to make the join possible. When there are multiple

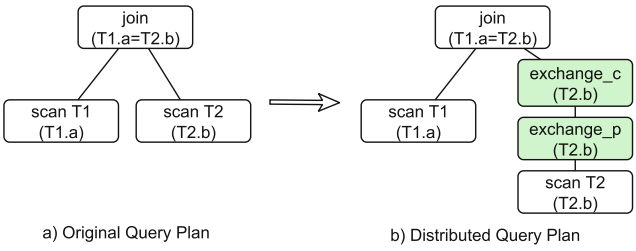

Fig. 2. Exchange operator allocation

places needing re-shuffling in a query plan, multiple pairs of exchange operators have to be allocated and data will be re-shuffled several times.

Operator Scheduling. As discussed above, a pair of exchange operators might be running on different sever nodes, and a consumer operator has to collect results from all working producer operators. A consumer operator has to wait if some of the producer operators have not yet finished their portion of assigned tasks. In order to coordinate all operators involved in such distributed computing tasks, a light-weight scheduler is deployed for synchronization of tasks, so that an operator will not be executed unless all input it requires has already been prepared. Before the intermediate results are collected by a consumer operator, they are temporarily stored in a local memory pool for better utilization of distributed memory storage. The scheduler module is duplicated across all servers, so that any one of the servers can be in charge of coordinating tasks. However, as for individual queries, DCODE activates only the scheduler on the server node accepting the query.

An example of operator scheduling in parallel query execution of a simple group-by query is shown in Fig. 3. We assume the group-by query is on a non-partitioning column. The scheduler module will then be activated on the server node accepting the query, i.e., Node 3 in Fig. 3. The sequence of function calls as well as data transferring flow are labeled as (1)–(9) in Fig. 3. In (1), the scheduler on Node 3 will first send an execution request (as a remote function call) to the producer operators `exchange_p` on all available server nodes that possess the column data, i.e., Node 1 and 2 in Fig. 3. In (2), each corresponding producer operator obtains the data through an access engine of column storage (as a local function call). This data will be stored in a local memory pool in (3). The producer operator will then notify the scheduler that the request has

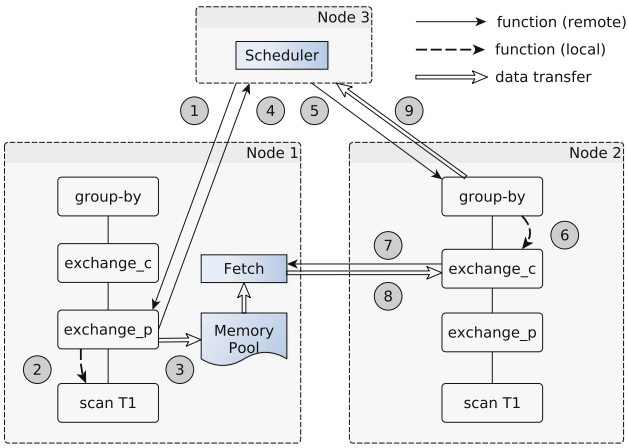

Fig. 3. Operator scheduling in DCODE

been completed in (4). The scheduler will wait till all execution requests are completed to begin next operation. In (5), another execution request is sent to the group-by operator on each server which starts the consumer operators `exchange_c` in (6). The consumer operator will call a remote *Fetch* function to obtain data from the memory pool in (7). The data will then be re-shuffled and sent back to a corresponding remote consumer operator in (8). In the end, after all nodes finish the group-by operator, the results will be transferred back to Node 3 in (9). Note that instead of collecting data directly from the consumer operator and doing all aggregation on Node 3, we apply the group-by operator to the intermediate results on each server node for better utilization of distributed computing resources.

2.3 Materialization Strategy

In a column database engine, although the intermediate computation could be based on column formatted data, the final results still need to be delivered as row formatted tuples. The operation that transforms columns into rows is called *materialization*. (1) An *early materialization* (EM) strategy [13] will start forming row formatted tuples as soon as query processing begins. (2) A *late materialization* (LM) strategy [8], on the contrary, will not start forming tuples until part of the query plan has been processed. The advantage of EM is that it reduces disk I/O cost for data access. As columns are added to tuples immediately when they are encountered, the column values are accessed only once during query processing. The advantage of LM is that it is more CPU efficient as it requires fewer tuples to be transformed from column formatted data into row formatted data during query processing. It also allows query operators to use high performance vectorized operations over the compressed value list of offset positions.

In a distributed column database engine, besides the aforementioned CPU and disk I/O cost, data re-shuffling incurs additional network transmission cost in the total cost of a query plan. We investigated three different strategies: an early materialization strategy, a late materialization strategy, and a *mixed materialization* strategy [11]. In an early materialization strategy, column values are stitched into row formatted tuples as soon as a column is encountered. After column values are added into intermediate results, they will be re-shuffled together with the re-shuffling column. The consequence of early materialization is that although CPU and disk I/O costs are reduced, network transmission cost is increased for re-shuffling more data. In a late materialization strategy, the offset positions are used in both processing intermediate results and re-shuffling. Although CPU and disk I/O costs increase, network transmission cost reduce. Since there exists a trade-off between early and late materialization, we further proposed a mixed materialization strategy to take advantage of both. When there are multiple operators accessing different columns in a distributed query plan, for columns whose early materialization would have higher network latency during re-shuffling steps than re-access latencies, it is better to delay their materialization; while for others, the conclusion will be the reverse.

2.4 Optimizations in Query Processing

Beyond the core techniques discussed above, we have identified some subtle or system-dependent optimization opportunities in query parsing, query execution, and multithreaded processing during the implementation of DCODE system.

Remote Compilation. During query processing, the input query text has to be first transformed into a query plan as an operator tree, and then transformed into a sequence of lower-level instructions accessing and operating column data. Such a process is called query compilation. Instead of compiling an entire query on a single server node and then sending generated code fragments to corresponding servers, it is more efficient to have all servers compile the same query simultaneously. Such remote compilation has two benefits: 1. No code transmission is needed. 2. The code generated locally can be optimized for execution based on different data characteristics and computing resources that are actually available, such as the number of CPU cores and the size of memory.

Operator Push-down. Push-down optimization is also supported for a number of operators such as group-by, order-by, and top-N. And it is not only for reducing the intermediate result size but also for reducing the network transmission cost in distributed query processing. However, these operators cannot be directly moved down in an operator tree of a distributed query plan. Instead, an equivalent query plan has to be derived by keeping additional information for these operators. For example, if a group-by operator has to be moved down through a pair of exchange operators, the additional information kept will depend on the type of aggregation function, such as `SUM()` and `AVG()`.

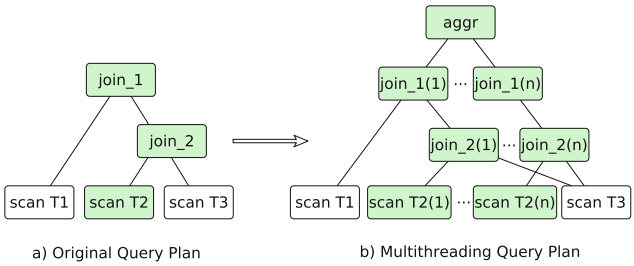

Fig. 4. Partitioning table selection for optimizing multithreading processing

Optimization for Multithreaded Processing. In modern DBMSs with multiple CPU cores, each thread is in charge of subtasks of computation with partial data. One key issue for gaining better performance using multithreading is load balancing. In the base open-source engine system of DCODE, a simple heuristic is employed to always partition the largest base table in a query plan, presumably dividing the largest computation task. However, as the size of intermediate

results also depends on the previous operations and the selectivity of corresponding operators, a larger base table does not guarantee larger intermediate results.

We adopt a cost-based optimizer for finding the best partitioning table to improve multithreaded processing efficiency [15]. The optimizer searches a space of all possible tables for partitioning, and ranks those partitioning tables based on a scoring function that captures multithreading efficiency. Once the optimal table for partitioning has been discovered, it generates a multithreaded execution plan. A concrete example of multithreaded processing is shown in Fig. 4. Assume there are three tables T1, T2, and T3 in the query. T2 first joins with T3, and then joins with T1 (Fig. 4a). If the cost-based optimizer determines that partitioning T2 will lead to the most efficient multithreading performance, it will split T2 into n parts with each part assigned to a separate thread (CPU core) for parallel processing. As shown in shaded region of Fig. 4b, each thread computes a partial join of T2 with T3 and then a partial join with T1 separately. The final result is obtained by aggregating results from all threads.

3 Implementation and Evaluation

3.1 Implementation

We developed the DCODE system based on MonetDB [10], a state-of-the-art open-source column-oriented database engine optimized for analytic query processing over large data sets. MonetDB revolutionized almost all layers of a database management system. Its innovative contributions include a vertical partitioning storage model, a hardware conscious query execution architecture, vectorized execution, jut-in-time light-weight compression, adaptive indexing, and database cracking. The good performance of the DCODE system is largely based on the outstanding query execution engine built inside MonetDB.

To achieve distributed query processing capability for a column database, we have added additional data structures and optimization algorithms in many core components of the execution engine of MonetDB. Specifically, we added functions for parsing query text for generating a distributed query execution plan. We defined new query operators such as an exchange operator and corresponding evaluation algorithms. We added optimization functions for improving query processing efficiency. We added new statements for function generation and new low-level instructions for query execution. In total, we have developed over 10K lines of code inside the MonetDB kernel, and over 20K line of debugging code for testing and verification.

3.2 Evaluation

We deployed both a physical Linux server cluster and a virtual Linux container (LXC) cluster in our lab environment. Each physical server node in our cluster is running an Ubuntu Linux 3.5.0-17-generic x86_64 operating system on 2x Intel(R) Xeon(R) E5-2680@2.70 GHz processors each with 16 effective threads and 20,480 KB cache. Each server node has 188 GB of RAM.

We used the standard TPC-H benchmark [5] to evaluate the performance of the DCODE system for analytic queries. We modified the DBGen program to generate distributed data by hash partitioning over a particular key column. The partitioning profile is stored locally so that each server node can access column data properly based on the distribution information. As discussed previously, because an optimal partitioning scheme is hard to achieve for distributed query processing, we simply partitioned the largest table in the TPC-H benchmark such as `LINEITEM` and `ORDERS` table in experiments although DCODE can handle arbitrary horizontal partitioning.

We evaluated parallel query performance of the DCODE system by comparison with other cutting-edge analytic query processing systems such as Impala v1.1.1 [3], Hive 0.10 [2] (in CDH 4.3.1), and Spark v1.0 [6]. Note that, these versions were the most up-to-date available when we completed our development, test and benchmarking work. For fairness, all systems are set up and evaluated in the same environment. For Impala and Spark, all tables were stored using its column format defined by Parquet [7].

In Fig. 5, a selection of parallel execution experiment results are shown for TPC-H queries Q1, Q3, Q4, Q6, Q8, Q14 and Q16 of scale factor *SF1000* on 16 server nodes. Missing results for a particular system means the query was not supported. From these and other results we concluded that the DCODE system has better coverage for TPC-H queries. As Fig. 5 shows, when using query latency as the yardstick, DCODE always outperforms the other systems. With Q1, Impala and Hive achieved similar performance results to DCODE. Both were orders of magnitude slower in executing the remaining queries. Hive also suffered from the overhead of its MapReduce framework. Spark 1.0 did not achieve comparable performance because the project was still in an early stage especially for query processing supports. We expect all these systems to have made improvements since we completed our study.

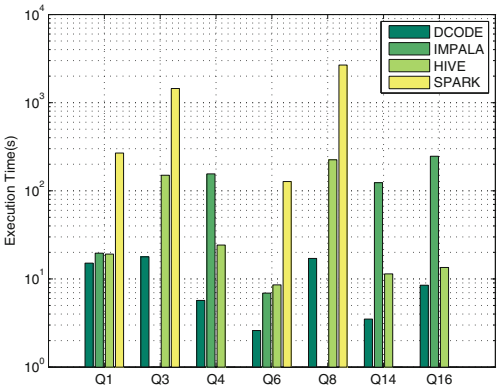

Fig. 5. Evaluation on TPC-H queries

4 Conclusion

In this paper, a novel distributed column-oriented database engine (DCODE) is proposed. The framework uses multiple techniques, e.g. data horizontal partitioning, exchange operator allocation, operator scheduling, materialization strategy selection, and some other optimizations for query processing, the details of which are introduced respectively. We measured the performance of our DCODE system using the TPC-H benchmark. We examined system effectiveness. Our system can automatic process queries in SQL which made comparison with existing advanced systems, such as HIVE, IMPALA and SPARK quite possible. Our results demonstrate that column orientation is ideal for analytic processing. We also demonstrate that with the right architecture and judicious selection of implementations, we can assemble just the right technology to enable parallel distributed query processing in an existing open-source database.

References

1. http://incubator.apache.org/drill/
2. https://archive.apache.org/dist/hive/hive-0.10.0/
3. https://impala.io/
4. http://prestodb.io/
5. http://www.tpc.org/tpch/
6. https://spark.apache.org/releases/spark-release-1-0-0.html/
7. http://parquet.incubator.apache.org/
8. Abadi, D.J., Myers, D.S., DeWitt, D.J., Madden, S.: Materialization strategies in a column-oriented DBMS. In: Proceedings of the 23rd International Conference on Data Engineering, Istanbul, Turkey, 15–20 April 2007, pp. 466–475 (2007)
9. Abouzeid, A., Bajda, K., Abadi, D., Silberschatz, A., Rasin, A.: Hadoopdb: an architectural hybrid of mapreduce and DBMS technologies for analytical workloads. Proc. VLDB Endow. **2**(1), 922–933 (2009)
10. Boncz, P.A., Zukowski, M., Nes, N.J.: MonetDB/X100: hyper-pipelining query execution. In: Proceedings of International Conference on Verly Large Data Bases (VLDB) 2005. Very Large Data Base Endowment (2005)
11. Ku, C., Liu, Y., Mortazavi, M., Cao, F., Chen, M., Shi, G.: Optimization strategies for column materialization in parallel execution of queries. In: Decker, H., Lhotská, L., Link, S., Spies, M., Wagner, R.R. (eds.) DEXA 2014, Part II. LNCS, vol. 8645, pp. 191–198. Springer, Heidelberg (2014)
12. Graefe, G.: Volcano - an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng. **6**(1), 120–135 (1994)
13. Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., Bear, C.: The vertica analytic database: C-store 7 years later. Proc. VLDB Endow. **5**(12), 1790–1801 (2012)
14. Lee, R., Luo, T., Huai, Y., Wang, F., He, Y., Zhang, X.: Ysmart: yet another sql-to-mapreduce translator. In: Proceedings of the 2011 31st International Conference on Distributed Computing Systems, Washington, DC, USA, pp. 25–36 (2011)
15. Liu, Y., Mortazavi, M., Cao, F., Chen, M., Shi, G.: Cost-based data-partitioning for intra-query parallelism. In: Proceedings of International Baltic Conference on DB and IS (2014)

16. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T.: Dremel: Interactive analysis of web-scale datasets. Proc. VLDB Endow. **3**(1–2), 330–339 (2010)
17. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: a column-oriented DBMS. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 553–564 (2005)
18. Zhou, J., Bruno, N., Lin, W.: Advanced partitioning techniques for massively distributed computation. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, pp. 13–24 (2012)