Large Scale On-Line Analytical Processing

Yifu Huang
School of Computer Science, Fudan University
Shanghai 200433, China
huangyifu@fudan.edu.cn

Abstract

There is a growing need for ad-hoc and routine analysis of extremely large data sets, especially at internet companies where innovation critically depends on being able to analyze terabytes of data collected every day. This task is referred to as large scale on-line analytical processing (OLAP). Traditional solutions of OLAP are data warehouses built by parallel database products, e.g., Teradata, but are usually prohibitively expensive at this scale. Besides, many of the people who analyze this data are entrenched procedural programmers, who find the declarative, SQL style to be unnatural. The success of the more procedural map-reduce programming model, and its associated scalable implementations on commodity hardware, is evidence of the above. The map-reduce paradigm is the modern solution to large scale data over cluster. However, it focuses on off-line analytical processing, is too low-level and rigid, and leads to a great deal of custom user code that is hard to maintain, and reuse. Large scale OLAP brings MapReduce-based systems into OLAP, thus achieves both scalability and efficiency. In this paper, I survey state-of-art systems of large scale OLAP, and study both design overview and implementation details.

1. Introduction

At a growing number of organizations, innovation revolves around the collection and analysis of enormous data sets such as web crawls, search logs, and click streams. Internet companies such as Amazon, Google, Microsoft, and Yahoo! are prime examples. Analysis of this data constitutes the innermost loop of the product improvement cycle. For example, the engineers who develop search engine ranking algorithms spend much of their time analyzing search logs looking for exploitable trends.

The sheer size of these data sets dictates that it be stored and processed on highly parallel systems, such as shared-nothing clusters. Parallel database products, e.g., Teradata, Oracle RAC, Netezza, offer a solution by providing a simple SQL query interface and hiding the complexity of the physical cluster. These products however, can be prohibitively expensive at web scale. Besides, they wrench programmers away from their preferred method of analyzing data, namely writing imperative scripts or code, toward writing declarative queries in SQL, which they often find unnatural, and overly restrictive.

As evidence of the above, programmers have been flocking to the more procedural map-reduce [6] programming model. A map-reduce program essentially performs a groupby-aggregation in parallel over a cluster of machines. The programmer provides a map function that dictates how the grouping is performed, and a reduce function that performs the aggregation. What is appealing to programmers about this model is that there are only two high-level declarative primitives (map and reduce) to enable parallel processing, but the rest of the code, i.e., the map and reduce functions, can be written in any programming language of choice, and without worrying about parallelism.

Unfortunately, the map-reduce model has its own set of limitations. Its one-input, two-stage data flow is extremely rigid. To perform tasks having a different data flow, e.g., joins or n stages, inelegant workarounds have to be devised. Also, custom code has to be written for even the most common operations, e.g., projection and filtering. These factors lead to code that is difficult to reuse and maintain, and in which the semantics of the analysis task are obscured. Moreover, the opaque nature of the map and reduce functions impedes the ability of the system to perform optimizations.

The task called Large scale OLAP aims at analyzing massive data sets with fast response. So it brings MapReduce-based systems into OLAP, thus achieves both scalability and efficiency. In this paper, I survey state-of-art systems of large scale OLAP which are designed and implemented by great universities and companies such as Yahoo!, Microsoft, Facebook, Yale and Google, and focus on their design overview and implementation details. I divided related 9 papers into three groups. Group I (Pig [8], SCOPE [5] and Hive [11, 12]) mainly trans-

lates SQL-like query languages into MapReduce-based jobs. Group 2 (HadoopDB [9, 1, 2, 3]) installs database on datanode. Group 3 (Dremel [7]) combines multi-level trees and columnar layout.

2. Group I: Pig, SCOPE and Hive

2.1. Pig

Apache Pig is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The salient property of Pig programs is that their structure is amenable to substantial parallelization, which in turns enables them to handle very large data sets. At the present time, Pig's infrastructure layer consists of a compiler that produces sequences of Map-Reduce programs, for which large-scale parallel implementations already exist (e.g., the Hadoop subproject). Pig's language layer currently consists of a textual language called Pig Latin, which has the following key properties:

Ease of programming. It is trivial to achieve parallel execution of simple, "embarrassingly parallel" data analysis tasks. Complex tasks comprised of multiple interrelated data transformations are explicitly encoded as data flow sequences, making them easy to write, understand, and maintain.

Optimization opportunities. The way in which tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.

Extensibility. Users can create their own functions to do special-purpose processing.

The authors from Yahoo! describe a new language called Pig Latin that they have designed to fit in a sweet spot between the declarative style of SQL, and the low-level, procedural style of map-reduce. The accompanying system, Pig, is fully implemented, and compiles Pig Latin into physical plans that are executed over Hadoop, an open-source, map-reduce implementation. They give a few examples of how engineers at Yahoo! are using Pig to dramatically reduce the time required for the development and execution of their data analysis tasks, compared to using Hadoop directly. They also report on a novel debugging environment that comes integrated with Pig, that can lead to even higher productivity gains. Pig is an open-source, Apache-incubator project, and available for general use.

The overarching design goal of Pig is to be appealing to experienced programmers for performing ad-hoc analysis of extremely large data sets. Consequently, Pig Latin has a number of features that might seem surprising when viewed from a traditional database and SQL perspective. In this section, we describe the features of Pig, and the rationale behind them.

Dataflow Language. In Pig Latin, a user specifies a sequence of steps where each step specifies only a single, highlevel data transformation. This is stylistically different from the SQL approach where the user specifies a set of declarative constraints that collectively define the result. While the SQL approach is good for non-programmers and/or small data sets, experienced programmers who must manipulate large data sets often prefer the Pig Latin approach.

Quick Start and Interoperability. Pig is designed to support ad-hoc data analysis. If a user has a data file obtained, say, from a dump of the search engine logs, she can run Pig Latin queries over it directly. She need only provide a function that gives Pig the ability to parse the content of the file into tuples. There is no need to go through a time-consuming data import process prior to running queries, as in conventional database management systems. Similarly, the output of a Pig program can be formatted in the manner of the users choosing, according to a user-provided function that converts tuples into a byte sequence. Hence it is easy to use the output of a Pig analysis session in a subsequent application, e.g., a visualization or spreadsheet application such as Excel.

Nested Data Model. Programmers often think in terms of nested data structures. Databases, on the other hand, allow only flat tables, i.e., only atomic fields as columns, unless one is willing to violate the First Normal Form (1NF). Pig Latin has a flexible, fully nested data model, and allows complex, non-atomic data types such as set, map, and tuple to occur as fields of a table.

UDFs as First-Class Citizens. To accommodate specialized data processing tasks, Pig Latin has extensive support for user-defined functions (UDFs). Essentially all aspects of processing in Pig Latin including grouping, filtering, joining, and per-tuple processing can be customized through the use of UDFs. The input and output of UDFs in Pig Latin follow our flexible, fully nested data model. Consequently, a UDF to be used in Pig Latin can take non-atomic parameters as input, and also output non-atomic values. This flexibility is often very useful as shown by the following example.

Parallelism Required. Since Pig Latin is geared toward processing web-scale data, it does not make sense to consider non-parallel evaluation. Consequently, we have only included in Pig Latin a small set of carefully chosen primitives that can be easily parallelized. Language primitives that do not lend themselves to efficient parallel evaluation (e.g., non-equi-joins, correlated subqueries) have been deliberately excluded. Such operations can of course, still be carried out by writing UDFs. However, since the language does not provide explicit primitives for such operations, users are aware of how efficient their programs will be and whether they will be parallelized.

Debugging Environment. In any language, getting a da-

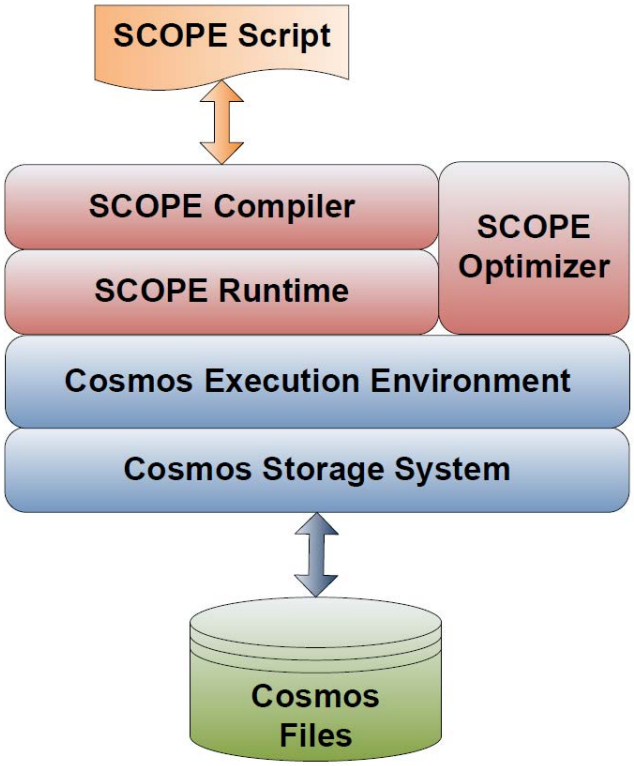

Figure 1. Cosmos Software Layers

ta processing program right usually takes many iterations, with the first few iterations usually having some user-introduced erroneous processing. At the scale of data that Pig is meant to process, a single iteration can take many minutes or hours (even with largescale parallel processing). Thus, the usual run-debug-run cycle can be very slow and inefficient. Pig comes with a novel interactive debugging environment that generates a concise example data table illustrating the output of each step of the users program. The example data is carefully chosen to resemble the real data as far as possible and also to fully illustrate the semantics of the program. Moreover, the example data is automatically adjusted as the program evolves.

2.2. SCOPE

Then SCOPE as shown in Fig. 1 is developed by Microsoft. In this paper, the author present a new scripting language, SCOPE (Struc-tured Computations Optimized for Parallel Execution), targeted for large-scale data analysis that is under development at Microsoft. Many users are familiar with relational data and SQL. SCOPE intentionally builds on this knowledge but with simplifi-cations suited for the new execution environment. Users familiar with SQL require little or no training to use SCOPE. Like SQL, data is modeled as sets of rows composed of typed columns. Every rowset has a well-defined schema. The SCOPE runtime provides implementations of many standard physical operators, saving users from implementing similar functionality repetitively. SCOPE is being used daily for a variety of data analysis and data mining applications inside Microsoft.

SCOPE is a declarative language. It allows users to focus on the data transformations required to solve the problem at hand and hides the complexity of the underlying platform and implementa-tion details. The SCOPE compiler and optimizer are responsible for generating an efficient execution plan and the runtime for executing the plan with minimal overhead.

SCOPE is highly extensible. Users can easily define their own functions and implement their own versions of operators: extractors (parsing and constructing rows from a file), processors (rowwise processing), reducers (groupwise processing), and combiners (combining rows from t-wo inputs). This flexibility greatly extends the scope of the language and allows users to solve problems that cannot be easily expressed in traditional SQL.

SCOPE provides functionality similar to views in SQL. This feature greatly enhances modularity and code reusability. It can also be used to restrict access to sensitive data.

SCOPE supports writing a program using traditional nested SQL expressions or as a series of simple data transformations. The latter style is often preferred by programmers who are used to thinking of a computation as a series of steps. We illustrate the usage of SCOPE by the following example.

The SCOPE scripting language resembles SQL but with C# ex-pressions. This design choice offers several advantages. Its re-semblance to SQL reduces the learning curve for users and eases porting of existing SQL scripts into S-COPE. SCOPE expressions can use C# libraries. Custom C# classes can compute functions of scalar values, or manipulate whole rowsets. A SCOPE script consists of a sequence of commands. Except for a few auxiliary commands, commands are data transformation operators that take one or more rowsets as input, perform some operation on the data, and output a rowset. Every rowset has a well-defined schema that all its rows must adhere to. By default, a command takes the result rowset of the previous command as input. As shown in Example 1, the output command in the first SCOPE script takes the result of the previous s-elect command as its input. SCOPE commands can also take named inputs and users can name the output of a command using as-signment. The output of a command can be used by subsequent commands one or more times. The second script in Example 1 shows an example of named inputs where e, s1, s2, s3 represent the result of the corresponding command. Named inputs/outputs enable users to write scripts in multiple (small) steps, a style pre-ferred by some programmers. SCOPE supports a variety of data types, including int, long, double, float, DateTime, string, bool and

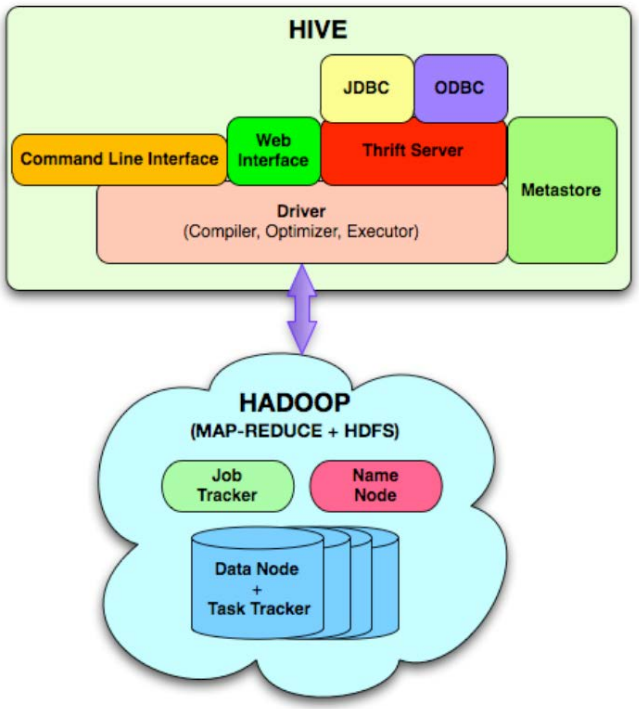

Figure 2. Hive System Architecture

their nullable counter-parts. SCOPE uses C# semantics for nulls, which differs from SQL null semantics. Null compares equal to null. Null compared to a non-null value is always false. Null sorts high. The aggregates ignore nulls in SCOPE. A script writer can view operators as being entirely serial; map-ping the script to an efficient parallel execution plan is handled completely by the SCOPE compiler and optimizer.

2.3. Hive

The Apache Hive data warehouse software facilitates querying and managing large datasets residing in distributed storage. Hive provides a mechanism to project structure onto this data and query the data using a SQL-like language called HiveQL. At the same time this language also allows traditional map/reduce programmers to plug in their custom mappers and reducers when it is inconvenient or inefficient to express this logic in HiveQL.

The Hive as shown in Fig. 2 in Group 1 is developed by Facebook. In this paper, the authors present Hive, an open-source data warehousing solution built on top of Hadoop. Hive supports queries expressed in a SQL-like declarative language - HiveQL, which are compiled into mapreduce jobs that are executed using Hadoop. In addition, HiveQL enables users to plug in custom map-reduce scripts into queries. The language includes a type system with support for tables containing primitive types, collections like arrays and maps, and nested compositions of the same. The underlying IO libraries can be extended to query data in custom formats. Hive also includes a system catalog - Metastore C that contains schemas and statistics, which are useful in data exploration, query optimization and query compilation. In Facebook, the Hive warehouse contains tens of thousands of tables and stores over 700TB of data and is being used extensively for both reporting and ad-hoc analyses by more than 200 users per month.

Data Model. Data in Hive is organized into: Tables - These are analogous to tables in relational databases. Each table has a corresponding HDFS directory. The data in a table is serialized and stored in files within that directory. Users can associate tables with the serialization format of the underlying data. Hive provides builtin serialization formats which exploit compression and lazy de-serialization. Users can also add support for new data formats by defining custom serialize and de-serialize methods (called SerDe's) written in Java. The serialization format of each table is stored in the system catalog and is automatically used by Hive during query compilation and execution. Hive also supports external tables on data stored in HDFS, NFS or local directories. Partitions - Each table can have one or more partitions which determine the distribution of data within subdirectories of the table directory. Suppose data for table T is in the directory /wh/T. If T is partitioned on columns ds and ctry, then data with a particular ds value 20090101 and ctry value US, will be stored in files within the directory /wh/T/ds=20090101/ctry=US. Buckets - Data in each partition may in turn be divided into buckets based on the hash of a column in the table. Each bucket is stored as a file in the partition directory.

Query Language. Hive provides a SQL-like query language called HiveQL which supports select, project, join, aggregate, union all and sub-queries in the from clause. HiveQL supports data definition (DDL) statements to create tables with specific serialization formats, and partitioning and bucketing columns. Users can load data from external sources and insert query results into Hive tables via the load and insert data manipulation (DML) statements respectively. HiveQL currently does not support updating and deleting rows in existing tables. HiveQL supports multi-table insert, where users can perform multiple queries on the same input data using a single HiveQL statement. Hive optimizes these queries by sharing the scan of the input data. HiveQL is also very extensible. It supports user defined column transformation (UDF) and aggregation (UDAF) functions implemented in Java. In addition, users can embed custom map-reduce scripts written in any language using a simple row-based streaming interface, i.e., read rows from standard input and write out rows to standard output. This flexibility does come at a cost of converting rows from and to strings.

Hive Architecture. The main components of Hive are: External Interfaces - Hive provides both user interfaces like

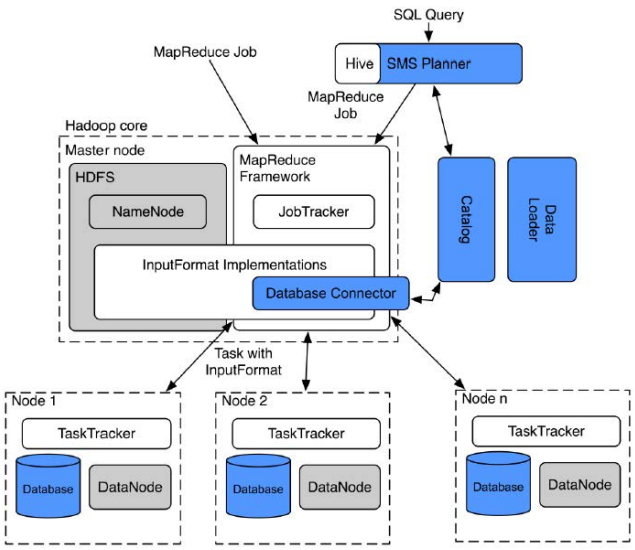

Figure 3. The HadoopDB Architecture

command line (CLI) and web UI, and application programming interfaces (API) like JDBC and ODBC. The Hive Thrift Server exposes a very simple client API to execute HiveQL statements. Thrift is a framework for cross-language services, where a server written in one language (like Java) can also support clients in other languages. The Thrift Hive clients generated in different languages are used to build common drivers like JDBC (java), ODBC (C++), and scripting drivers written in php, perl, python etc. The Metastore is the system catalog. All other components of Hive interact with the metastore. The Driver manages the life cycle of a HiveQL statement during compilation, optimization and execution. On receiving the HiveQL statement, from the thrift server or other interfaces, it creates a session handle which is later used to keep track of statistics like execution time, number of output rows, etc. The Compiler is invoked by the driver upon receiving a HiveQL statement. The compiler translates this statement into a plan which consists of a DAG of mapreduce jobs. The driver submits the individual map-reduce jobs from the DAG to the Execution Engine in a topological order. Hive currently uses Hadoop as its execution engine.

3. Group II: HadoopDB

The HadoopDB as shown in Fig. 3 is developed by Yale. There tend to be two schools of thought regarding what technology to use for data analysis in such an environment. Proponents of parallel databases argue that the strong emphasis on performance and efficiency of parallel databases makes them well-suited to perform such analysis. On the other hand, others argue that MapReduce-based systems are better suited due to their superior scalability, fault tolerance, and flexibility to handle unstructured data. In this paper, the authors explore the feasibility of building a hybrid system that takes the best features from both technologies; the prototype they built approaches parallel databases in performance and efficiency, yet still yields the scalability, fault tolerance, and flexibility of MapReduce-based systems.

Performance. Performance is the primary characteristic that commercial database systems use to distinguish themselves from other solutions, with marketing literature often filled with claims that a particular solution is many times faster than the competition. A factor of ten can make a big difference in the amount, quality, and depth of analysis a system can do. High performance systems can also sometimes result in cost savings. Upgrading to a faster software product can allow a corporation to delay a costly hardware upgrade, or avoid buying additional compute nodes as an application continues to scale. On public cloud computing platforms, pricing is structured in a way such that one pays only for what one uses, so the vendor price increases linearly with the requisite storage, network bandwidth, and compute power. Hence, if data analysis software product A requires an order of magnitude more compute units than data analysis software product B to perform the same task, then product A will cost (approximately) an order of magnitude more than B. Efficient software has a direct effect on the bottom line.

Fault Tolerance. Fault tolerance in the context of analytical data workloads is measured differently than fault tolerance in the context of transactional workloads. For transactional workloads, a fault tolerant DBMS can recover from a failure without losing any data or updates from recently committed transactions, and in the context of distributed databases, can successfully commit transactions and make progress on a workload even in the face of worker node failures. For read-only queries in analytical workloads, there are neither write transactions to commit, nor updates to lose upon node failure. Hence, a fault tolerant analytical DBMS is simply one that does not have to restart a query if one of the nodes involved in query processing fails. Given the proven operational benefits and resource consumption savings of using cheap, unreliable commodity hardware to build a shared-nothing cluster of machines, and the trend towards extremely low-end hardware in data centers, the probability of a node failure occurring during query processing is increasing rapidly. This problem only gets worse at scale: the larger the amount of data that needs to be accessed for analytical queries, the more nodes are required to participate in query processing. This further increases the probability of at least one node failing during query execution. Google, for example, reports an average of 1.2 failures per analysis job. If a query must restart each time a node fails, then long, complex queries are difficult to complete.

Ability to run in a heterogeneous environment. As de-

scribed above, there is a strong trend towards increasing the number of nodes that participate in query execution. It is nearly impossible to get homogeneous performance across hundreds or thousands of compute nodes, even if each node runs on identical hardware or on an identical virtual machine. Part failures that do not cause complete node failure, but result in degraded hardware performance become more common at scale. Individual node disk fragmentation and software configuration errors can also cause degraded performance on some nodes. Concurrent queries (or, in some cases, concurrent processes) further reduce the homogeneity of cluster performance. On virtualized machines, concurrent activities performed by different virtual machines located on the same physical machine can cause 2-4If the amount of work needed to execute a query is equally divided among the nodes in a shared-nothing cluster, then there is a danger that the time to complete the query will be approximately equal to time for the slowest compute node to complete its assigned task. A node with degraded performance would thus have a disproportionate effect on total query time. A system designed to run in a heterogeneous environment must take appropriate measures to prevent this from occurring.

Flexible query interface. There are a variety of customer-facing business intelligence tools that work with database software and aid in the visualization, query generation, result dash-boarding, and advanced data analysis. These tools are an important part of the analytical data management picture since business analysts are often not technically advanced and do not feel comfortable interfacing with the database software directly. Business Intelligence tools typically connect to databases using ODBC or JDBC, so databases that want to work with these tools must accept SQL queries through these interfaces. Ideally, the data analysis system should also have a robust mechanism for allowing the user to write user defined functions (UDFs) and queries that utilize UDFs should automatically be parallelized across the processing nodes in the shared-nothing cluster. Thus, both SQL and non-SQL interface languages are desirable.

4. Group III: Dremel

The Dremel as shown in Fig. 4 is developed by Google. Dremel is a scalable, interactive ad-hoc query system for analysis of read-only nested data. By combining multi-level execution trees and columnar data layout, it is capable of running aggregation queries over trillion-row tables in seconds. The system scales to thousands of CPUs and petabytes of data, and has thousands of users at Google. In this paper, we describe the architecture and implementation of Dremel, and explain how it complements MapReduce-based computing. We present a novel columnar storage representation for nested records and discuss experiments

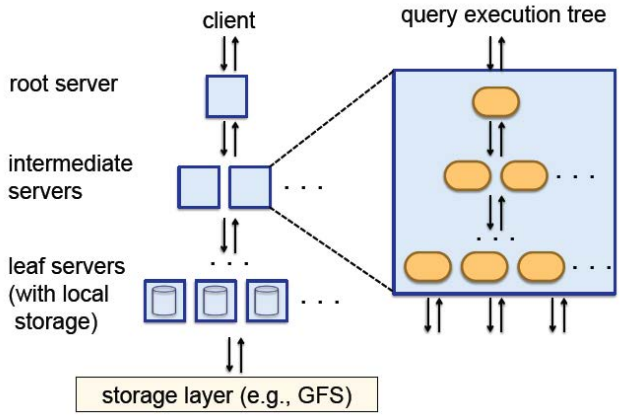

Figure 4. System architecture and execution inside a server node

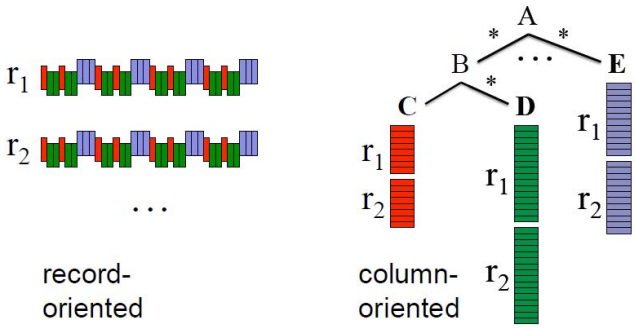

Figure 5. Record-wise vs. columnar representation of nested data

on few-thousand node instances of the system.

Data Model. The data model originated in the context of distributed systems (which explains its name, Protocol Buffers), is used widely at Google, and is available as an open source implementation. The data model is based on strongly-typed nested records. The nested data model backs a platform-neutral, extensible mechanism for serializing structured data at Google. Code generation tools produce bindings for programming languages such as C++ or Java. Cross-language interoperability is achieved using a standard binary on-the-wire representation of records, in which field values are laid out sequentially as they occur in the record. This way, a MR program written in Java can consume records from a data source exposed via a C++ library. Thus, if records are stored in a columnar representation, assembling them fast is important for interoperation with MR and other data processing tools.

Nested Columnar Storage. Values alone do not convey the structure of a record. Given two values of a repeated field, we do not know at what level the value repeated (e.g., whether these values are from two different records, or two repeated values in the same record). Likewise, given a missing optional field, we do not know which enclosing records

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Figure 6. Column-striped representation of the sample data in Figure 2, showing repetition levels (r) and definition levels (d)

were defined explicitly. We therefore introduce the concepts of repetition and definition levels, which are defined below. Above we presented an encoding of the record structure in a columnar format. The next challenge we address is how to produce column stripes with repetition and definition levels efficiently. The algorithm recurses into the record structure and computes the levels for each field value. As illustrated earlier, repetition and definition levels may need to be computed even if field values are missing. Many datasets used at Google are sparse; it is not uncommon to have a schema with thousands of fields, only a hundred of which are used in a given record. Hence, we try to process missing fields as cheaply as possible. To produce column stripes, we create a tree of field writers, whose structure matches the field hierarchy in the schema. The basic idea is to update field writers only when they have their own data, and not try to propagate parent state down the tree unless absolutely necessary. To do that, child writers inherit the levels from their parents. A child writer synchronizes to its parents levels whenever a new value is added.

5. Conclusion and Future Work

For future work, we can further bring in-memory techniques into large scale OLAP to future boost processing performance. An in-memory database (IMDB; also main memory database system or MMDB or memory resident database) is a database management system that primarily relies on main memory for computer data storage. It is contrasted with database management systems that employ a disk storage mechanism. Main memory databases are faster than disk-optimized databases since the internal optimization algorithms are simpler and execute fewer CPU instructions. Accessing data in memory eliminates seek time when querying the data, which provides faster and more predictable performance than disk. Applications where response time is critical, such as those running telecommunications network equipment and mobile advertising networks, often use main-memory databases. IMDBs have gained a lot of traction, especially in the data analytics space, starting in the mid-2000s - mainly due to cheaper RAM. With the introduction of non-volatile random access memory technology, in-memory databases will be able to run at full speed and maintain data in the event of power failure. Specifically, thinking about integrating in-memory column store MonetDB [4] and in-memory row store VoltDB [10] into Hadoop-based systems. Increase data volume of benchmark to find the bottleneck of in-memory database. And bring optimization techniques such as partition used in parallel databases to resolve bottleneck of in-memory database.

References

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[2] A. Abouzied, K. Bajda-Pawlikowski, J. Huang, D. J. Abadi, and A. Silberschatz. Hadoopdb in action: building real world applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 1111–1114, 2010.

[3] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1165–1176, 2011.

[4] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[5] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.

[7] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1099–1110, 2008.

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

SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009, pages 165–178, 2009.

[10] M. Stonebraker and A. Weisberg. The voltdb main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[11] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[12] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005, 2010.