Future Generation Computer Systems 63 (2016) 148–156

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Benchmarking performance for migrating a relational application to a parallel implementation

Krishna Karthik Gadiraju, Manik Verma, Karen C. Davis *, Paul G. Talaga

EECS Department, University of Cincinnati, Cincinnati, OH 45221-0030, USA

ARTICLE INFO

Article history:
Received 31 May 2015
Received in revised form
19 December 2015
Accepted 22 December 2015
Available online 7 January 2016

Keywords:
Hive
Hadoop
Benchmarking
Big data
SQL
Queries

ABSTRACT

Many organizations rely on relational database platforms for OLAP-style querying (aggregation and filtering) for small to medium size applications. We investigate the impact of scaling up the data sizes for such queries. We intend to illustrate what kind of performance results an organization could expect should they migrate current applications to big data environments. This paper benchmarks the performance of Hive (Thusoo et al., 2009) [9], a parallel data warehouse platform that is a part of the Hadoop software stack. We set up a 4-node Hadoop cluster using Hortonworks HDP 1.3.2 (Hortonworks HDP 1.3.2). We use the data generator provided by the TPC-DS benchmark (DSGen v1.1.0) to generate data of different scales. We compare the performance of loading data and querying for SQL and Hive Query Language (HiveQL) on a relational database installation (MySQL) and on a Hive cluster, respectively. We measure the speedup for query execution for three dataset sizes resulting from the scale up. Hive loads the large datasets faster than MySQL, while it is marginally slower than MySQL when loading the smaller datasets. Query execution in Hive is also faster. We also investigate executing Hive queries concurrently in workloads and conclude that serial execution of queries is a much better practice for clusters with limited resources.

1. Introduction

Big data refers to petabyte scale datasets that cannot be managed and analyzed using traditional database management systems and data warehouses. The last decade has seen an enormous rise in the size of data collected by different organizations. According to Baru et al. [1], studies have indicated that enterprise data is estimated to grow from 0.5 ZB in 2008 to 35 ZB in 2020. Traditional relational database systems are considered incapable of handling data of such scale. Industry blogs [2–4] indicate that Hive has advantages over relational databases, but typically there is no experimental evidence to support or quantify the discussion. Moreover, it is not always clear to a smaller organization what performance gains they can expect to achieve for their application by migrating it to a big data platform with a modest investment in additional hardware. One such organization approached us with this question. After analyzing their current configuration and typical queries, we selected a hardware/software configuration and test query to enable us to provide some indication of what they could expect if they scaled up their data. Their application only runs one data intensive query at a time, and they wished to own their own hardware, so our first phase of experiments target this scenario. There are no other results in the literature that address this question.

Our second phase of experimentation investigates performance for workloads consisting of multiple copies of queries running simultaneously. While this is a typical practice in relational environments, which are optimized for transaction processing, our experiments illustrate that it is more efficient to run Hive queries one at a time rather in a concurrent workload. Previous benchmarking studies use smaller datasets than our experiments, and some do not consider analytical queries or load times; others do not compare their results to a relational implementation. None of the previous studies consider a workload executing concurrently.

In this paper, we discuss the features of Apache Hive and compare its performance against a relational database system. We use a query and data that are a part of the TPC-DS [5] benchmarking standard. In the following sections, we briefly describe the features of Hive. We present our experimental setup, procedures, and results obtained for loading and querying both MySQL and Hive. In our second phase of experiments we investigate the performance of Hive under a workload setting, where multiple queries are running concurrently, as is typical in relational database applications. In contrast, all previous studies of Hive performance execute a single query at a time.

* Corresponding author. Tel.: +1 513 556 2214; fax: +1 513 556 7326.
E-mail address: karen.davis@uc.edu (K.C. Davis).

http://dx.doi.org/10.1016/j.future.2015.12.015
0167-739X/

2. Apache Hive

Apache Hive is a distributed data warehouse that is a part of the Hadoop software stack. Hive provides SQL-like query and analysis capability for large datasets, but is primarily a service on top of the Hadoop File System, not a full database system [6]. The Hive table definitions and mapping to the data are stored in the Metastore. The Metastore manages the data definitions and mappings to the data and provides interface services. The default Metastore is Apache Derby, which is limited to only one connection at a time. Using MySQL, Postgres, Oracle, or SQL Server for the Hive Metastore supports multiple concurrent Hive sessions [7].

Queries written in HiveQL are translated into a series of MapReduce [8] jobs. The HiveQL query is translated into a DAG (Directed Acyclic Graph) of MapReduce jobs. MapReduce is a parallel programming framework. The MapReduce jobs defined in the DAG are executed in parallel on the different nodes in the cluster where the data is stored to obtain the results. A typical Hive installation [9] has a Metastore, a Thrift server, which provides a client API for executing the HiveQL statements, external interfaces such as command line interface (CLI), a driver, which is responsible for management of the life cycle of a HiveQL statement, and a compiler which is used to translate a HiveQL statement it receives from the driver into a DAG of MapReduce jobs. Once the DAG of MapReduce jobs is prepared by the compiler, the driver invokes the execution engine (which in the case of Hive is Hadoop) to execute the MapReduce jobs. A Hive data model consists of tables, partitions and buckets. A Hive table is similar to a table in a relational database and is made up of rows and columns. Each time a table is created in Hive, a new directory is created on HDFS (Hadoop Distributed File System) and data related to that table is stored in that folder. A table can have one or more partitions. Each partition is stored as a separate directory within the table directory and the data is stored in whichever partition directory it belongs to. The partitions can be further sub-divided into buckets, depending upon the hash of a column in the table. The buckets are stored as individual files within their respective partition directories.

3. Initial load and query execution: experimental setup

3.1. Hardware configuration

A 4-node Hadoop cluster was set up using Hortonworks HDP 1.3.2 [10]. We were limited to a small cluster because of the feasibility of an academic setup wherein we used the available departmental hardware. Each machine has dual quad-core Intel Xeon processors for a total of 16 hyper-threaded cores per machine, 48 GB RAM and 3.08 TB HDD. All four machines communicate with each other using a gateway machine. The MySQL machine shared the same processor and RAM, but had a 2.05 TB HDD installed. The six machines mentioned here are connected together using a Cisco SG 200-26 26-Port Gigabit Smart Switch.

3.2. Software configuration

The Hadoop cluster was set up using Hortonworks HDP 1.3.2. The version of Hive used in this study is 0.11, and the version of MySQL used is 5.1.71. Centos 6.4 minimal operating system was used to set up the Hadoop cluster, which ran Hadoop version 1.2.

3.3. Experimental procedure

There are several benchmarking standards defined to benchmark the performance of Hadoop such as Sorting programs (Hadoop Sort Program [11], TeraSort [12]), GridMix [13] and HiBench Benchmarking Suite [14], but none of them have well-defined queries or a schema necessary for evaluating the run time performance of a big data management system such as Hive. BigBench [15] and Hive Performance Benchmark [16] both define a schema and dataset for benchmarking Hive, but BigBench is based on the TPC-DS benchmark and provides a larger variety and scale of datasets and queries. While the structured part of BigBench is based on TPC-DS, it also adds several semi-structured and unstructured data components [15]. Since we are dealing with how Hive performs against a relational system, we use the TPC-DS benchmark to analyze the performance of Hive.

TPC-DS (Transaction Processing Performance Council–Decision Support) is a benchmark for evaluating decision support systems. It defines 99 distinct queries that serve a typical business analysis environment [17,5]. TPC-DS uses a snowstorm schema, which is a collection of several snowflake schemas. The schema has been created to model the decision support functions of a retail product supplier. We selected Query 7 from the TPC-DS benchmark as a representative OLAP-style query. It joins 5 tables and contains 4 aggregation operations, 1 group by operation, and 1 order by operation. We modified the original version of the query to remove the "top 100" expression in order to focus on aggregation and filtering over a fact table and several dimension tables. The modified and HiveQL versions of the query are as shown below.

The modified SQL query is:

```
select i_item_id,
  avg(ss_quantity) agg1,
  avg(ss_list_price) agg2,
  avg(ss_coupon_amt) agg3,
  avg(ss_sales_price) agg4
from store_sales, customer_demographics,
  date_dim, item, promotion
where ss_sold_date_sk = d_date_sk and
  ss_item_sk = i_item_sk and
  ss_cdemo_sk = cd_demo_sk and
  ss_promo_sk = p_promo_sk and
  cd_gender = 'F' and
  cd_marital_status = 'D' and
  cd_education_status = 'College' and
  (p_channel_email = 'N' or p_channel_event = 'N')
    and
  d_year = 2001
group by i_item_id
order by i_item_id;
```

Since HiveQL uses joins rather than listing tables using the ',' operator in the FROM clause as in the TPC-DS query above, the revised query is shown below:

```
select i_item_id,
  avg(ss_quantity) agg1,
  avg(ss_list_price) agg2,
  avg(ss_coupon_amt) agg3,
  avg(ss_sales_price) agg4
from store_sales ss join date_dim d on
  (ss.ss_sold_date_sk = d.d_date_sk)
join item i on (ss.ss_item_sk = i.i_item_sk)
join promotion p on (ss.ss_promo_sk =
  p.p_promo_sk)
join customer_demographics cd on
  (ss.ss_cdemo_sk = cd.cd_demo_sk)
  where
  cd_gender = 'F' and
  cd_marital_status = 'D' and
  cd_education_status = 'College' and
  (p_channel_email = 'N' or p_channel_event = 'N')
```