2017 IEEE 33rd International Conference on Data Engineering

BigBench V2: The New and Improved BigBench

Ahmad Ghazal* Todor Ivanov† Pekka Kostamaa‖ Alain Crolotte‡
Ryan Voong§ Mohammed Al-Kateb‡ Waleed Ghazal ¶ Roberto V. Zicari†
* Futurewei Technologies Inc., Santa Clara, CA, USA
Email: ahmad.ghazal@huawei.com
† Frankfurt Big Data Lab, Goethe University Frankfurt, Frankfurt am Main, Germany
Email: {todor, zicari}@dbis.cs.uni-frankfurt.de
‖ OpenX, Culver City, CA, USA
Email: {pekka.kostamaa}@openx.com
‡ Teradata Labs, El Segundo, CA, USA
Email: {alain.crolotte, mohammed.al-kateb}@teradata.com
§ University of California, Los Angeles, CA, USA
Email: ryanvoong@ucla.edu
¶ Redondo Union High School, Redondo Beach, CA, USA
waleedghazal99@gmail.com

***Abstract*—Benchmarking Big Data solutions has been gaining a lot of attention from research and industry. BigBench is one of the most popular benchmarks in this area which was adopted by the TPC as TPCx-BB. BigBench, however, has key shortcomings. The structured component of the data model is the same as the TPC-DS data model which is a complex snowflake-like schema. This is contrary to the simple star schema Big Data models in real life. BigBench also treats the semi-structured web-logs more or less as a structured table. In real life, web-logs are modeled as key-value pairs with unknown schema. Specific keys are captured at query time - a process referred to as late binding. In addition, eleven (out of thirty) of the BigBench queries are TPC-DS queries. These queries are complex SQL applied on the structured part of the data model which again is not typical of Big Data workloads. In this paper[1], we present BigBench V2 to address the aforementioned limitations of the original BigBench. BigBench V2 is completely independent of TPC-DS with a new data model and an overhauled workload. The new data model has a simple structured data model. Web-logs are modeled as key-value pairs with a substantial and variable number of keys. BigBench V2 mandates late binding by requiring query processing to be done directly on key-value web-logs rather than a pre-parsed form of it. A new scale factor-based data generator is implemented to produce structured tables, key-value semi-structured web-logs, and unstructured data. We implemented and executed BigBench V2 on Hive. Our proof of concept shows the feasibility of BigBench V2 and outlines different ways of implementing late binding.**

I. Introduction

The problem of storing and analyzing Big Data in its structured, semi-structured, and non-structured forms continues to be of major interest to academic research and industrial products. Several commercial proprietary (e.g., Aster [1] and Cloudera [2]) and open source systems (e.g., Presto [3] and Spark [4]) were developed in the past few years to tackle the challenges and grasp the opportunities of Big Data.

As Big Data systems expand and mature, there is a need for benchmarks that help assessing their functionality and performance. So far, BigBench [5] is arguably the only benchmark that provides an end-to-end solution for Big Data benchmarking. It was adopted - without major design or architectural changes - by the TPC [6]. However, BigBench has major shortcomings due to its reliance on TPC-DS [7] and its simplistic handling of semi-structured data. This paper aims at addressing these shortcomings by a proposal for BigBench V2. BigBench V2 is self-contained, independent of TPC-DS, and more representative of real life Big Data systems. For the rest of this paper, we refer to the original BigBench and its TPC implementation as BigBench and we use BigBench V2 as the name of the new improved benchmark. Before describing BigBench V2, we first elaborate on the limitations of BigBench.

BigBench conveniently re-used components of TPC-DS to fill in the data model and data generation of the structured part of the benchmark. Also, eleven TPC-DS queries are used in BigBench to cover some of the retail analytics described in McKinsey's report [8]. TPC-DS is a benchmark for decision support and has a complex snowflake-like data model. TPC-DS workload is based on complex SQL constructs with lots of joins, aggregations, and sub-queries. The complex data model and queries in TPC-DS are not representative of Big Data systems and applications with simple schemas which also imply fewer joins and sub-queries.

The main limitation of BigBench is in the way it handles web-logs (i.e., semi-structured data). It handles web-logs as a structured table and all queries are processed against a fixed schema. This is contrary to real life applications, in which web-logs consist of a large and unknown set of keys that makes it impractical to parse these web-logs and create a schema out of them upfront. The practical approach in these cases is to extract the keys (i.e., columns) required to satisfy each query at run-time. This technique of looking up the structure of data at run-time is known as *late binding* [9] [10].

BigBench V2 separates from TPC-DS with a simple data model. The new data model still has the variety of structured, semi-structured, and unstructured data as the original

[1]Part of this work was done while Ahmad Ghazal was at Oracle, and Pekka Kostamaa and Ryan Voong were at Teradata.

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.167

IEEE computer society

BigBench data model. The difference is that the structured part has only six tables that capture necessary information about users (customers), products, web pages, stores, online sales and store sales. We developed a scale factor-based data generator for the new data model. The web-logs are produced as *key-value* pairs with two sets of keys. The first set is a small set of keys that represent fields from the structured tables like IDs of users, products, and web pages. The other set of keys is larger and is produced randomly. This set is used to simulate the real life cases of large keys in web-logs that may not be used in actual queries. Product reviews are produced and linked to users and products as in BigBench but the review text is produced synthetically contrary to the Markov chain model [11] used in BigBench. We decided to generate product reviews in this way because the Markov chain model requires real data sets which limits our options for products and makes the generator hard to scale.

For the workload queries, all 11 TPC-DS queries on the complex structured part are removed and replaced by simpler queries mostly against the key-value web-logs. The new BigBench V2 queries have only 5 queries on the structured part versus 18 in BigBench. This change has no impact on the coverage of the different business categories done in BigBench. In addition to the removal of TPC-DS queries, BigBench V2 mandates late binding [12] but it does not impose a specific implementation of it. This requirement means that a system using BigBench V2 can extract the keys and their corresponding values per query at run-time. Other than the changes above, BigBench V2 is the same as BigBench including metric definition and computation.

The remainder of this paper is organized as follows. Section II covers work related to Big Data benchmarking. Section III describes the new simplified data model. Our scalable custom-made data generator is discussed in section IV. BigBench V2 workload queries and late binding requirements are outlined in section V. Section VI presents our proof of concept for BigBench V2 using Hive. Finally, Section VII summarizes the paper and suggests future directions.

II. Related Work

Quite a few benchmarks have been proposed and developed recently to measure the performance and applicability of Big Data systems and applications [13]. Among these different benchmarks, BigBench [5] [14] is arguably the first concrete piece of work towards benchmarking Big Data. BigBench added semi-structured and unstructured data to TPC-DS [7] and provided 30 queries on Big Data retail analytics per the McKinsey's report [8]. The work in [15] implemented BigBench in Hadoop and developed BigBench queries using HiveQL.

Most of the other Big Data benchmarks focus on particular applications or domains. HiBench [16] and SparkBench [17] [18] are micro-benchmark suites developed specifically to stress test the capabilities of Hadoop (both MapReduce and HDFS) and Spark systems using many separate workloads. Likewise, MRBS [19] provides workloads of five different domains with the focus on evaluating the dependability of MapReduce systems. CloudSuite [20] and CloudRank-D [21] are benchmark suites tailored for cloud systems. Both consist of multiple scale-out workloads that test a diverse set of cloud system functionality. CloudSuite focuses on identifying processor micro-architecture and memory system inefficiencies, whereas CloudRank-D stresses the data processing capabilities of cloud systems similar to HiBench. LinkBench [22] is a benchmark, developed by Facebook, using synthetic social graph to emulate social graph workload on top of databases such as MySQL. BigFUN [23] is another benchmark that is based on a social network use case with synthetic semi-structured data in JSON format. The benchmark focuses exclusively on micro-operation level. The benchmark workload consists of queries with various operations such as simple retrieves, range scans, aggregations, joins, as well as inserts and updates.

More generic benchmarks include BigFrame [24], PRIMEBALL [25], and BigDataBench [26]. BigFrame offers the ability to create a benchmark customized to a specific set of data and workload requirements. PRIMEBALL [25] includes various use cases involving both queries and batch processing on different types of data. BigDataBench [26] is yet another effort that proposes a benchmark suite for variety of workloads and datasets in order to address a wider range of Big Data applications. While BigDataBench, addresses the semi-structured data in the data model, it does not take late binding into consideration as a key concept for applications dealing with semi-structured data.

A recent SPEC Big Data Research Group survey [27] provided a summary of the existing Big Data benchmarks and those that are currently under development. The study reviewed the aforementioned benchmarks as well as other benchmarks by outlining their characteristics with the goal of helping both researchers and practitioners choose the appropriate benchmark for their needs. Comparing and contrasting these benchmarks to BigBench, we identify the uniqueness and superiority of BigBench [5] as follows:

- BigBench is technology agnostic, whereas many of the existing benchmarks are technology or component bound (HiBench [16], SparkBench [17], MRBS [19], CloudSuite [20], LinkBench [22], and PigMix [28]).
- BigBench addresses the data variety (structured, semi-structured and unstructured data), which is not the case with most current Big Data benchmarks (PigMix [28], CALDA [29], and TPCx-HS [30]).
- BigBench is an end-to-end benchmark with a unified data model that covers all important types of Big Data analytics (30 queries), unlike the micro-benchmark suites that consist of many separate domain specific workloads (HiBench [16], SparkBench [17], CloudSuite [20] and CloudRank-D [21]).

Based on the above advantages of BigBench over other benchmarks, the TPC chose it as a standard for Big Data benchmarking and named it TPCx-BB [6]. However, as discussed in the Introduction section, BigBench has limitations in terms of being dependent on TPC-DS and its simplistic and unrealistic handling of semi-structured data. This paper proposes enhancements of BigBench through BigBench V2 that aims at fixing these limitations. The original TPCx-BB

based on BigBench can be enhanced using BigBench V2 as well.

III. Data Model

BigBench V2 data model is a simple custom-made model representing user activities on an online retail store as shown in Figure 1. The data model meets the new workload queries described in section V and covers the variety of the data needed in Big Data.

The structured part of the model consists of six tables with their full schemas shown in Table I. The *user* table captures data (name, state, country, etc.) for all registered users as well as users who visit the brick and mortar and online stores. Products offered by the retailer are stored in the *product* table. The *product* table has product name, its description, category, class information, price, and the lowest competitor price. The retailer online pages are described in the *webpage* table that has webpage URL, description, and type. The *websale* table stores sales information including customer/user who purchased a product, which product was sold, product quantity, and the date and time of the transaction. The *storesale* table is similar to the *websale* table with an additional field for store name. The directed arrows on Figure 1 indicate primary-foreign key relationship between the different tables. For example, *websale* has a many-to-one relationship with the *product* and the *user* tables. Note that the six structured tables are covered in BigBench with a bigger and more complex schema. For example, BigBench structured part (from TPC-DS) has separate tables for date and time, while BigBench V2 just uses a simple timestamp field to represent date and time. Also, BigBench V2 folded product categories into the *product* table while BigBench has separate tables for product categories and classes.

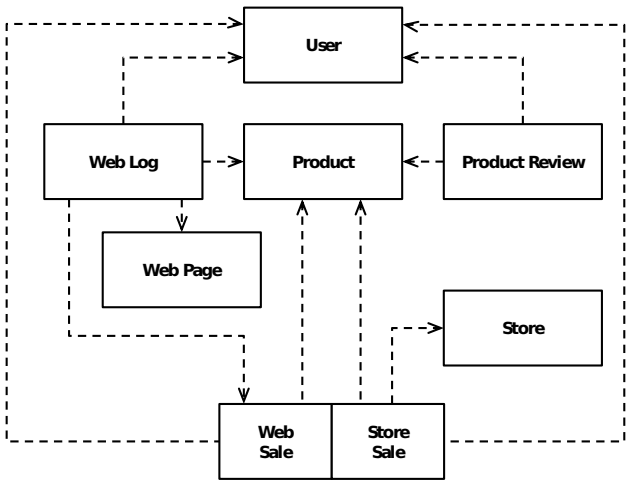

Fig. 1: Data model

The semi-structured component is represented by web-logs capturing user clicks just like BigBench. However, unlike BigBench, the web-log entries are in the form of key-value pairs with no relational schema. It logs the activities (i.e., clicks) of a user while the user visits different webpages, handles shopping carts, or checkout products. These actions generate keys and values related to the *user*, *product* and *webpage* tables. Another set of random keys and their values are augmented to each web-log entry to represent real life scenarios where web-logs have large number of unknown keys. Section IV explains how these two sets are generated with our new data generator.

TABLE I: Schema of the six structured tables

Table	Columns
user	u_user_id u_name
product	p_id p_name p_category_id p_category_name p_price
webpage	w_web_page_id w_web_page_name w_web_page_type
websale	ws_transaction_id ws_user_id ws_product_id ws_quantity ws_timestamp
storesale	ss_transaction_id ss_store_id ss_user_id ss_product_id ss_quantity ss_timestamp
store	s_store_id s_store_name

An example of a click (i.e., one entry in web-logs) is shown below. In this example a user $user_1$ at time t_1 clicked on a webpage w_1 that has information about product p_1. The click has additional 100 random keys along with their values to simulate the large number of key-value pairs in real life web-logs.

<**user**,$user_1$> <**time**,t_1> <**webpage**,w_1> <**product**,p_1>
<**key1**,*value1*> <**key2**,*value2*> ... <**key100**,*value100*>

The third component of the data model is the unstructured product reviews text. Similar to BigBench, product reviews are represented as a table with a wide text field to hold the reviews as shown in Table II. In addition to the review text, the table captures the user who did the review along with the product on which the review was submitted. It also has the user's overall rating of the product.

IV. Data Generation

The data generator of BigBench V2 is a scalable synthetic data generator that meets the design and requirements of the data model described in Section III. The data generator is based on a cardinality scale factor, similar to the way data

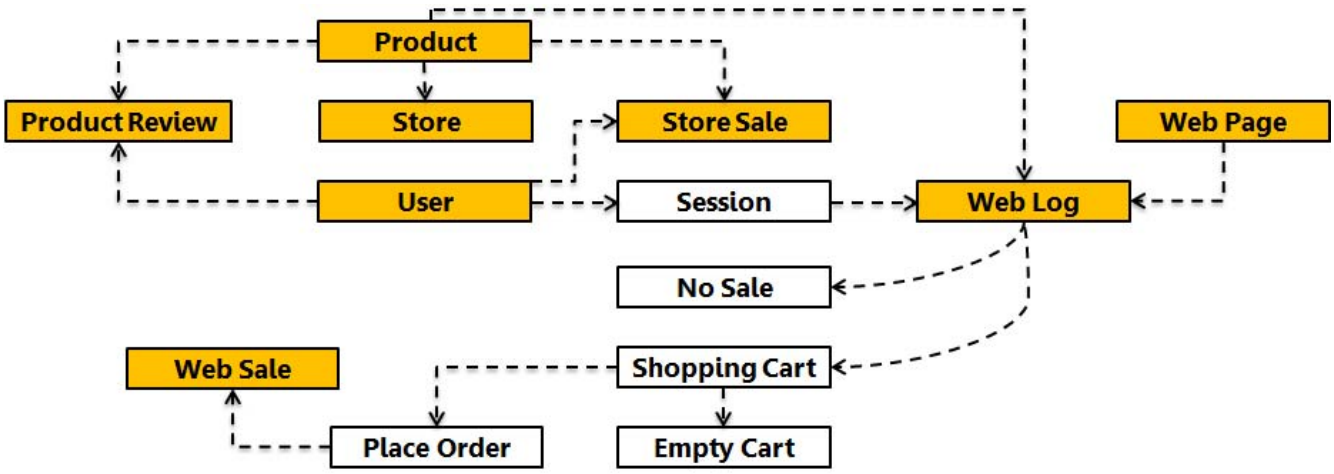

Fig. 2: Data generation process flow

TABLE II: Schema of product review table.

Table	Columns
productreview	pr_review_id pr_product_id pr_rating pr_content

is generated in TPC benchmarks. The data generation covers all the six relational tables: *product*, *webpage*, *user*, *store*, *storesale*, and *websale*. In addition, the data generator produces the key-value *weblog* and unstructured *productreview* in sync with the structured tables.

The cardinalities of *user*, *storesale*, *websale* and *weblogs* grow linearly with the scale factor. The *product* table is scaled sublinearly since in real life the number of new products does not grow proportional to the number of users. Data in the *webpage* table is assumed to be static and does not grow with the scale factor. Table III shows cardinality for some example scale factors. This data illustrates the linear, sub-linear, and constant growth of the different data sources. The exact cardinality formula for the different tables and web-logs will be included in a detailed and extended report with the full specifications of the benchmark.

TABLE III: Cardinality for various scaling factors

Data\Factor	1	100	1000	10000
webpage	26	26	26	26
product	1,000	1,900	4,063	10,900
user	10,900	109,900	1,009,900	10,009,900
store	100	105	150	600
web sale	143,880	1,450,680	13,330,680	132,130,680
store sale	59,950	604,450	5,554,450	55,054,450
product review	163,863	1,652,163	15,182,163	150,482,163
weblog	23,000,000	236,000,000	2,200,000,000	21,500,000,000

The data generator logic flow is shown in Figure 2. Boxes with colored fill represent final data sets (six tables, web-logs, and product reviews). The directed arrows indicate a primary-foreign key relationship between the different data stores. For example, *websale* has a many-to-one relationship with the *product* table. The data generator first produces the data for *user*, *product*, and *webpage* tables based on the scale factor. The entries in *weblogs* are produced in correlation to the data of *user*, *product*, and *webpage* tables. As shown in Section V, web-logs analytics examine the user clicks in terms of sessions (i.e., sequence of clicks during a fixed amount of time). Some sessions lead to adding products to shopping carts and some are simply used for browsing. Sessions that involve shopping carts could be abandoned or may eventually lead to an actual sale. The data generator is driven by these different scenarios as illustrated in Figure 2. Actual sales are also captured in the *websale* table. Data generation for the *storesale* table is done independently but in sync with the data generated for other related tables.

As mentioned before, in real life, clicks typically have large and unknown number of keys. The keys are broken down into two sets. One set includes keys that are normally needed by queries such as price, timestamp, etc. The other set is simply for arbitrary keys with random values to support the large and unknown key requirements of late binding. Our data generator produces the web-logs data in JSON format since it is a commonly used format for key-value data.

In BigBench, the unstructured product reviews text generation is based on a Markov chain process using real life product reviews with limited number of categories. This method is not scalable since it is impossible to find real reviews on demand to support higher values of the scale factor. To address this issue, BigBench V2 data generator produces product reviews synthetically in conjunction with the *user* and *product* tables. The cardinality of these reviews grows linearly with the scale factor similar to the *user* table.

The relationships between the different boxes in Figure 2 (e.g., the average number of sessions per user, the average number of clicks per session, the percentage of clicks leading to a shopping cart, etc.) are captured in a configuration file and can be adjusted to control the data generation. The average number of random key-value pairs added to clicks is also captured in the configuration file. We plan to offer public

access to the binaries of the data generator along with a detailed data model.

V. WORKLOAD

One of the enhancements in BigBench V2 over BigBench is de-emphasizing the structured part and increasing the share of semi-structured data in the workload. This is accomplished by replacing all 11 complex TPC-DS queries that involve neither semi-structured nor unstructured part of the data model. The new queries are developed to mainly go after the semi-structured part of the new model. Queries 20 and 24 are also replaced since the data model does not have sales return. The remaining 17 queries are superficially rewritten to reflect new table and column names. BigBench V2 new queries answer the business questions listed below and their HiveQL code is included in the Appendix. To simplify referencing and tracking queries, we kept the query numbers of the original 17 BigBench queries as is. The new 13 queries re-use the numbers of the deleted queries which are : Q_5, Q_6, Q_7, Q_9, Q_{13}, Q_{14}, Q_{16}, Q_{17}, Q_{19}, Q_{20}, Q_{21}, Q_{22}, and Q_{23}.

- Q_5 : Find the 10 most browsed products.
- Q_6 : Find the 5 most browsed products that are not purchased.
- Q_7 : List users with more than 10 sessions. A session is defined as a 10-minute window of clicks by a user.
- Q_9 : Find the average number of sessions per registered user per month. Display the top ten users.
- Q_{13} : Find the average amount of time a user spends on the website.
- Q_{14} : Compare the average number of products purchased by users from one year to the next.
- Q_{16} : Find the top ten pages visited.
- Q_{17} : Find the top ten pages visited on a certain day (such as Valentine's Day).
- Q_{19} : Find out the days with the highest page views.
- Q_{20} : Do a user segmentation based on their preferred shopping method (online vs. in-store).
- Q_{21} : Find the most popular web page paths that lead to a purchase.
- Q_{22} : Show the number of unique visitors per day.
- Q_{23} : Show the users with the most visits.

The final 30 BigBench V2 queries pretty much answer the same type of business questions covered in the original BigBench. Table IV summarizes the business questions breakdown of the queries in BigBench V2 and BigBench side by side. More background information about the source of business questions in BigBench and BigBench V2 can be found in [5] and [8]. The table also shows the technical breakdown in terms of data source and types of queries. In terms of query processing, BigBench V2 has emphasis on the combination of procedural and declarative more than declarative and procedural alone like the case in BigBench. The majority of queries (60%) in BigBench were applied on the structured part on the expense of the semi-structured and unstructured data. In retail business, semi-structured data (capturing online user experiences and interactions) is normally more important than product reviews. On that basis, we applied most of the new queries in BigBench V2 on the web-logs.

TABLE IV: Technical and business query breakdown

Business Category	**BigBench**		**BigBench V2**	
	No. of queries	Percentage	No. of queries	Percentage
Marketing	18	60.0%	20	69.0%
Merchandising	5	16.7%	3	10.3%
Operations	4	13.3%	2	6.9%
Supply chain	2	6.77%	1	3.3%
New business models	1	3.3%	4	13.8%
Query Type	**BigBench**		**BigBench V2**	
	No. of queries	Percentage	No. of queries	Percentage
Declarative	10	33.3%	7	24.1%
Procedural	7	23.3%	4	13.3%
Declarative & Procedural	13	43.3%	19	65.6%
Data Source	**BigBench**		**BigBench V2**	
	No. of queries	Percentage	No. of queries	Percentage
Structured	18	60.0%	5	16.7%
Semi-Structured	7	23.3%	20	66.7%
Unstructured	5	16.7%	5	16.7%

One of the key contributions in BigBench V2 is mandating late binding. Web-logs cannot be accessed as a table by the workload queries and upfront parsing of the web-logs is not allowed. Only at run time, on a query by query basis, the system conducting the benchmark can know the keys needed from the web-logs.

There are different methods for implementing late binding. BigBench V2 does not require any specific one. In a high level, pulling keys at run time can be done through non-streaming and streaming methods. On one hand, non-streaming methods scan all records/entries of the web-logs, extract the keys, and make them available (for instance, through a table) to the rest of the query execution. Streaming methods, on the other hand, perform key extraction one record at a time (buffering can be used as an optimization) and the result is passed to the execution engine as a tuple/row. For example, if the web-logs are involved in a join then streaming provides one row at a time for the join execution in a data flow fashion. Streaming provides more parallelism and less memory requirements. However, materialized results from non-streaming methods can be re-used across different queries. Parsing web-logs (streaming or non-streaming) can be done natively by the Big Data software solution or can be done through an external tool. For example, SparkSQL and Drill have native support for JSON and can parse web-logs directly. In contrast, Hive needs an internal or external user-defined function (UDF) to parse web-logs. Section VI provides concrete examples of these different options done through our experiments.

VI. Proof of Concept

Similar to BigBench, BigBench V2 is technology agnostic and can be implemented on different engines. The official TPCx-BB [6], which is based on BigBench, is implemented using HiveQL with Hive [31] being the most commonly used data warehouse engine on Hadoop. We implemented BigBench V2 in Hive as well and developed queries using HiveQL. Section VI-A describes our experimental setup and Section VI-B discusses the implementation of the proof of concept. The actual experiments of the 30 queries and their results are discussed in section VI-C. Finally, section VI-D shows experiments of 3 queries on SparkSQL and Drill to illustrate different ways late binding can be applied.

A. Experimental Setup

We performed all BigBench V2 experiments, presented in the following sections (VI-C and VI-D), on our experimental system. This section gives a brief description of our test cluster.

Hardware: We use a dedicated cluster consisting of 4 nodes connected directly through a 1GBit Netgear switch. All 4 nodes are Dell PowerEdge T420 servers. The master node is equipped with 2x Intel Xeon E5-2420 (1.9GHz) CPUs - each with 6 cores, 32GB of main memory, and 1TB hard drive. The 3 worker nodes are equipped with 1x Intel Xeon E5-2420 (2.20GHz) CPU with 6 cores, 32GB of RAM and, 4x 1TB (SATA, 7.2K RPM, 64MB Cache) hard drives.

Software: The Ubuntu Server 14.04.1 LTS was installed on all 4 nodes, allocating the entire first disk. The Cloudera Distribution of Hadoop (CDH) versions 5.5.1 with Hive 1.1.0 were used in all experiments. The total storage capacity of the cluster is 13TB of which 8TB are effectively available as HDFS space. Due to resource limitations (only 3 worker nodes) of our setup, the cluster was configured to work with replication factor of two.

Data Generation and Loading: We ran our new BigBench V2 data generator with scale factor set to 1. The generated data files corresponding to each BigBench V2 table for scale factor (SF) 1 are outlined in Table V. Using a HiveQL script, we created the data model schema and loaded the 6 structured tables, the product reviews and the external web-logs table in Hive. The data loading times per table are also provided in Table V.

TABLE V: Data size and loading time

	Scale Factor 1	
Table Name	**Data Size**	**Loading Time (sec.)**
user	420 KB	16.633
product	84 KB	13.162
product review	6.44 MB	15.196
web log	19.7 GB	0.13
web page	4 KB	13.051
web sale	10.1 MB	16.104
store sale	10.4 MB	15.031
store	8 KB	13.878
Total:	19.8 GB	103.185

B. Implementation

The 6 structured tables and product reviews are defined as Hive tables. The HiveQL definition for the table *user* is shown below as an example. The field delimiter defines how the fields are separated in the text file, generated by the data generator. The location attribute describes the physical location of the HDFS file.

```
DROP TABLE IF EXISTS user;
CREATE TABLE user
( u_user_id             bigint,
  u_name                string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
STORED AS TEXTFILE
LOCATION 'hdfsDataPath/user';
```

The web-logs are produced in JSON format as mentioned in section IV and we capture it in HDFS as a file called *clicks.json*. We define an external table web-logs with a single text field that holds the JSON data. The definition of the Hive table web-logs is shown below.

```
CREATE EXTERNAL TABLE IF NOT EXISTS
web_logs (line string)
ROW FORMAT DELIMITED LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION 'hdfsPath/web_logs/clicks.json';
```

As mentioned in Section V, there are multiple ways to implement late binding and in Hive this can be done using internal or external UDFs. The purpose of this proof of concept, however, is to show that BigBench V2 queries can be easily implemented in Hive regardless which implementation is more efficient. In our Hive implementation we used the internal *json_tuple* user-defined table function. It accesses the external *web_logs* table with the help of *lateral_view_syntax* [32] and extracts keys from the JSON records. For example, Q_{16}, defined in Section V, uses the *json_tuple* UDTF to extract only the *wl_webpage_name* key from each JSON record:

```
Q16 HiveQL:
      select
          wl_webpage_name,
          count(*) as cnt
      from
          web_logs
              lateral view
              json_tuple (
              web_logs.line,
              'wl_webpage_name'
          ) logs as wl_webpage_name
      where
          wl_webpage_name is not null
      group by wl_webpage_name
      order by cnt desc
      limit 10;
```

There are other alternative internal UDFs like the *get_json_object* UDF that can be used for parsing of JSON records in Hive. An external way to access JSON files can be implemented using Hive Streaming in combination with Python scripts.

BigBench and TPCx-BB used Python scripts to implement the procedural constructs needed in the workload. The most

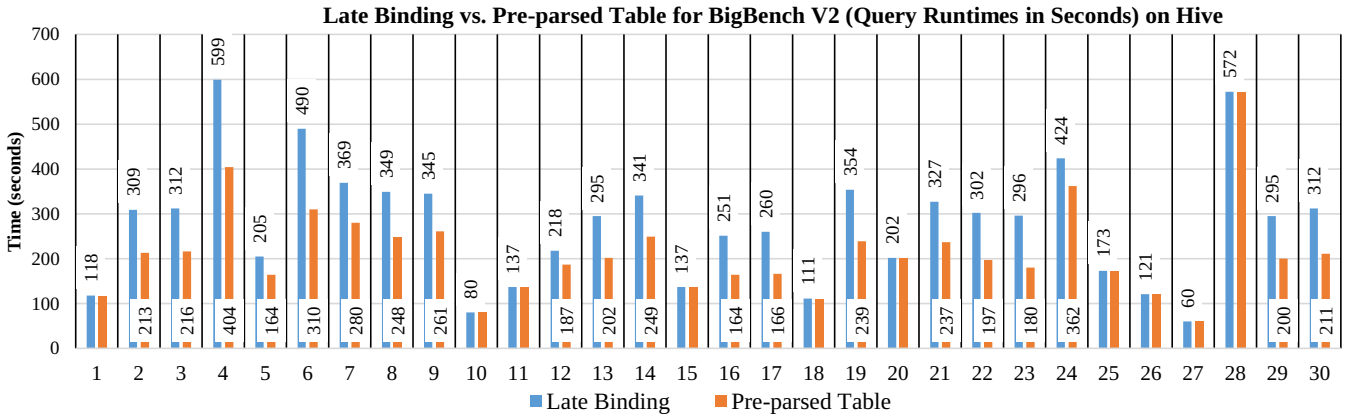

Fig. 3: BigBench V2 Hive results for SF1

common procedural constructs are: *sessionize* which identifies user sessions and *path* which performs path analysis. Using Python scripts is not only inefficient as an external function to Hive but is also complex since each usage of *sessionize* or *path* requires a custom written script. To avoid this complexity in BigBench V2 new queries, we implemented *sessionize* and *path* with native Hive UDF functions. We used these new and general UDFs in all relevant queries (new and old) of BigBench V2.

C. Hive Experiments

Apache Hive [31] [33] is the engine in which we implemented all 30 BigBench V2 queries and performed experiments with scale factors (SF) 1. The execution times (in seconds) of all queries are shown in Figure 3 with label *Late Binding*. Note that this data set is good enough as a proof of concept and further experiments with bigger scale factors are planned as future work. Overall, execution times vary which shows different complexities of these queries.

To investigate the overhead of late binding, we tried some of the queries to access a table that represents a pre-parsed form of the JSON data. Our data generator has an option to generate a table for web-logs which we used to produce a Hive table called *pre-parsed_web_logs*. The execution time of the modified queries for SF1 is labeled *Pre-parsed Table* and also shown in Figure 3. Note that, Q_1, Q_{10}, Q_{11}, Q_{15}, Q_{18}, ,Q_{20}, Q_{25}, Q_{26}, Q_{27}, and Q_{28} do not involve late binding. The execution time of these queries is the same with and without late bidning. Additionally, Table VI shows the execution time in seconds for the queries using the late binding approach and the respective modified queries using the web-log pre-parsed table. The difference in execution time illustrates the actual late binding overhead. The overhead ranges between 17% and 64% depending on the query and is on average around 43%. The total execution time of all queries with late binding is around 140 minutes and around 107 minutes with pre-parsed web-log table, which implies an overall overhead of around 23%.

D. Other Engines

In addition to our proof of concepts on Hive, we looked at other popular Big Data engines in order to study different ways of implementing late binding. For this purpose, we looked at three queries (Q_{16}, Q_{22}, and Q_{23}) in SparkSQL and Drill. Our comparison does not try to find which engine is better. Rather, it shows different alternatives of implementing late binding and handling key-value semi-structured data.

Apache Spark [4] [34] has became a popular alternative to the MapReduce framework, promising faster processing and offering advanced analytical capabilities by SparkSQL [35]. It natively supports HiveQL and can directly access the Hive metastore. This allowed us to execute three Hive queries (Q_{16}, Q_{22}, and Q_{23}) without any modifications. Using the latest Spark version 2.0.0, the queries were run with SF1 on the pre-loaded Hive metastore. Similar to the Hive experiments, we executed both the late binding and the pre-parsed web-log table HiveQL implementations of the three queries. The execution times are provided in Table VII and shown in Figure 4. The difference between implementations is labeled as *Overhead*. It ranges between 54% and 66%, which turns out to be very similar to the Hive overhead. For Q_{22} and Q_{23}, both Hive and Spark achieve the same late binding overhead. As mentioned in Section V, SparkSQL offers native JSON support. It can automatically infer the JSON schema through the use of the *org.apache.spark.sql.json* library, in which case no UDFs are required. However, for the sake of simplicity we leave this internal SparkSQL comparison for a future study.

Apache Drill [36] [37] is a columnar, schema-free SQL query engine that uses a JSON data model to enable queries on complex and nested data stored in Hadoop, NoSQL, or cloud storage. In comparison to Hive and SparkSQL, which rely on MapReduce and Spark for the data processing, Drill has its own optimizer that automatically restructures a query plan to leverage its internal processing capabilities. Using Drill version 1.7.0 installed on all four cluster nodes, we executed the same three queries. Unlike SparkSQL, Drill does not support HiveQL and we implemented the queries using the Drill's native JSON support (Section V). Query implementations were

TABLE VI: Late binding vs. pre-parsed table Hive for SF1

Query	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{12}	Q_{13}	Q_{14}	Q_{16}	Q_{17}	Q_{19}	Q_{21}	Q_{22}	Q_{23}	Q_{24}	Q_{29}	Q_{30}
Late binding (sec)	309	312	599	205	490	369	349	345	218	295	341	251	260	354	327	302	296	424	295	312
Pre-parsed table (sec)	213	216	404	164	310	280	248	261	187	202	249	164	166	239	237	197	180	362	200	211
Overhead (sec)	96	96	195	41	180	89	101	84	31	93	92	87	94	115	90	105	116	62	95	101
Overhead %	45	44	48	25	58	32	41	32	**17**	46	37	53	57	48	38	53	**64**	**17**	48	48

TABLE VII: Late binding overhead for other engines

Engine	SparkSQL			Drill		
Query	Q_{16}	Q_{22}	Q_{23}	Q_{16}	Q_{22}	Q_{23}
Late binding (sec)	232	234	231	223	249	266
Pre-parsed table (sec)	140	152	142	66	67	80
Overhead %	66	54	63	238	272	233

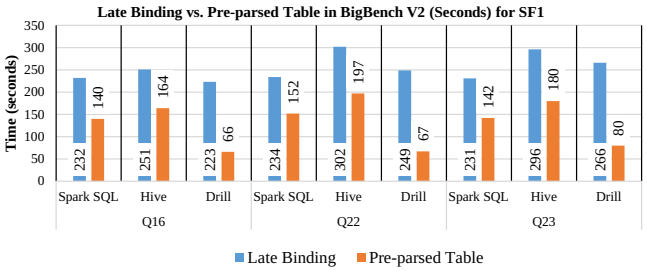

Fig. 4: BigBench V2 engines results for SF1

tested using late binding and pre-parsed Hive table. In fact, the only difference between the two implementations is in the *from* statement. In the case of late binding, it directly references the JSON file in HDFS. For the pre-parsed Hive table, it references the table in Hive as shown below for Q_{16}.

```
Q16 Drill:
 select
    wl_webpage_name,
    count(*) as cnt
 from
    /* using late binding */
    hdfs.`/hdfs_path/clicks.json`
    /* using pre-parsed Hive table */
    /* hive.bigbench.`pre-parsed_web_logs` */
 where
    wl_webpage_name is not null
 group by wl_webpage_name
 order by cnt desc
 limit 10;
```

The execution times on Drill are shown on Table VII and Figure 4. Interestingly, Drill performs very similar to SparkSQL in the late binding experiments, whereas it is 2-3 times faster than SparkSQL for the pre-parsed table queries. This results in a much greater overhead between 233% and 272% caused by the run-time parsing of the JSON file. In other words, the late binding overhead in Drill is almost 4 times bigger than the one observed in Hive and SparkSQL.

In summary, our proof of concept work proves that BigBench V2 is easy to implement as a self-contained benchmark with all required components. The benchmark is executed fully on Hive and partially on SparkSQL and Drill. The experiments on these three systems illustrate varying methods of late binding implementations and their corresponding overhead.

VII. Conclusions & Future Work

In this paper, we presented BigBench V2 - a major rework of BigBench data model and generator. The new data model and its corresponding generator reflect real life Big Data simple data models and late binding requirement. We built a custom-made and scale factor-based data generator for all components of the data model. All 11 TPC-DS queries are removed from BigBench and replaced with new queries in BigBench V2. These new queries answer similar business questions, but focus on analytics on the semi-structured web-logs. We also implemented a rigorous and a complete proof of concept on Hive. The proof of concept illustrates the feasibility and self containment of the benchmark. It also highlights the cost of late binding and how that varies among different engines. We hope these results can be useful for providers to enhance their respective engines to efficiently implement late binding.

We plan to make the data generator and queries available for the public. We also plan to propose enhancing TPCx-BB using BigBench V2 and work on the necessary changes for the specification and final queries written in HiveQL. Such an extension to the TPCx-BB should be straightforward since TPCx-BB is already based on HiveQL.

References

[1] Aster, www.teradata.com/teradata-aster/.

[2] Cloudera, www.cloudera.com.

[3] Presto, www.prestodb.io.

[4] Spark, www.spark.apache.org.

[5] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen, "BigBench: Towards An Industry Standard Benchmark for Big Data Analytics," in *SIGMOD*, 2013, pp. 1197–1208.

[6] TPCx-BB, www.tpc.org/tpcx-bb/default.asp.

[7] TPC-DS, www.tpc.org/tpcds/default.asp.

[8] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big Data: The Next Frontier for Innovation, Competition, and Productivity," 2011.

[9] B. Wooden and J. Coates, "Building A Common Information Model(CIM) Compliant Technical Add-on (TA)," 2014.

[10] Z. H. Liu, B. C. Hammerschmidt, D. McMahon, Y. Liu, and H. J. Chang, "Closing the Functional and Performance Gap between SQL and NoSQL," in *SIGMOD*, 2016, pp. 227–238.

[11] S. Meyn and R. L. Tweedie, *Markov Chains and Stochastic Stability*, 2nd ed. New York, NY, USA: Cambridge University Press, 2009.

[12] S. Brobst, "The Importance of Late Binding for Big Data Analytics," in *Extremely Large Databases Conference*, 2013.

[13] J. Zhan, R. Han, and R. V. Zicari, Eds., *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, ser. Lecture Notes in Computer Science, vol. 9495. Springer, 2016.

[14] T. Rabl, A. Ghazal, M. Hu, A. Crolotte, F. Raab, M. Poess, and H. Jacobsen, "BigBench Specification V0.1 - BigBench: An Industry Standard Benchmark for Big Data Analytics," in *Proceedings of the 2012 Workshop on Big Data Benchmarking*, 2012, pp. 164–201.

[15] B. Chowdhury, T. Rabl, P. Saadatpanah, J. Du, and H. Jacobsen, "A BigBench Implementation in The Hadoop Ecosystem," in *Proceedings of the 2013 Workshop on Big Data Benchmarking*, 2013, pp. 3–18.

[16] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench Benchmark Suite: Characterization of The MapReduce-based Data Analysis," in *Workshops Proceedings of the 26th IEEE ICDE International Conference on Data Engineering*, 2010, pp. 41–51.

[17] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "SparkBench: A Comprehensive Benchmarking Suite for In Memory Data Analytic Platform Spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015, pp. 53:1–53:8.

[18] D. Agrawal, A. R. Butt, K. Doshi, J. Larriba-Pey, M. Li, F. R. Reiss, F. Raab, B. Schiefer, T. Suzumura, and Y. Xia, "SparkBench - A Spark Performance Testing Suite," in *TPCTC*, 2015, pp. 26–44.

[19] A. Sangroya, D. Serrano, and S. Bouchenak, "MRBS: Towards Dependability Benchmarking for Hadoop MapReduce," in *Euro-Par: Parallel Processing Workshops*, 2012, pp. 3–12.

[20] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing The Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2012, pp. 37–48.

[21] C. Luo, J. Zhan, Z. Jia, L. Wang, G. Lu, L. Zhang, C. Xu, and N. Sun, "CloudRank-D: Benchmarking and Ranking Cloud Computing Systems for Data Processing Applications," *Frontiers of Computer Science*, vol. 6, no. 4, pp. 347–362, 2012.

[22] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: A Database Benchmark Based on The Facebook Social Graph," in *SIGMOD*, 2013, pp. 1185–1196.

[23] P. Pirzadeh, M. J. Carey, and T. Westmann, "BigFUN: A Performance Study of Big Data Management System Functionality," in *2015 IEEE International Conference on Big Data*, 2015, pp. 507–514.

[24] BigFrame, https://github.com/bigframeteam/BigFrame/wiki.

[25] J. Ferrarons, M. Adhana, C. Colmenares, S. Pietrowska, F. Bentayeb, and J. Darmont, "PRIMEBALL: A Parallel Processing Framework Benchmark for Big Data Applications in the Cloud," in *TPCTC*, 2013, pp. 109–124.

[26] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A Big Data Benchmark Suite from Internet Services," in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014*, 2014, pp. 488–499.

[27] T. Ivanov, T. Rabl, M. Poess, A. Queralt, J. Poelman, N. Poggi, and J. Buell, "Big Data Benchmark Compendium," in *TPCTC*, 2015, pp. 135–155.

[28] PigMix, https://cwiki.apache.org/confluence/display/PIG/PigMix.

[29] 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 *SIGMOD*, 2009, pp. 165–178.

[30] TPCx-HS, www.tpc.org/tpcx-hs/default.asp.

[31] Hive, hive.apache.org.

[32] Hive Lateral View, www.cwiki.apache.org/confluence/display/Hive/LanguageManual+LateralView.

[33] 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*, vol. 2, no. 2, pp. 1626–1629, 2009.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2012, pp. 15–28.

[35] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in *SIGMOD*, 2015, pp. 1383–1394.

[36] Drill, drill.apache.org.

[37] M. Hausenblas and J. Nadeau, "Apache drill: Interactive Ad-hoc Analysis at Scale," *Big Data*, vol. 1, no. 2, pp. 100–104, 2013.

APPENDIX
NEW QUERIES

- Q_5 :

```
select
    i_name,
    count(*) as cnt
from
    web_pages,
    product,
    (select
        js.wl_user_id,
        js.wl_product_id,
        js.wl_webpage_name
     from web_logs
        lateral view
            json_tuple
            (web_logs.line,
            'wl_user_id', 'wl_product_id',
            'wl_webpage_name'
            ) js as
        wl_user_id, wl_product_id,
        wl_webpage_name
     where
        js.wl_user_id is not null
        and js.wl_product_id is not null
      ) logs
where
    logs.wl_webpage_name = w_web_page_name
    and w_web_page_type = 'product look up'
    and logs.wl_product_id = i_product_id
group by i_name
order by cnt desc
limit 10;
```

- Q_6 :

```
drop view if exists browsed;

create view browsed as
select
    wl_product_id as br_id,
    count(*) as br_count
from
    web_pages,
    (select
        js.wl_user_id,
        js.wl_product_id,
        js.wl_webpage_name
    from web_logs
        lateral view
            json_tuple
            (web_logs.line,
            'wl_user_id', 'wl_product_id',
            'wl_webpage_name'
            ) js as wl_user_id,
            wl_product_id, wl_webpage_name
     where
        js.wl_user_id is not null
        and js.wl_product_id is not null
     ) logs
where
    wl_webpage_name = w_web_page_name
    and w_web_page_type = 'product look up'
group by wl_product_id;

drop view if exists purchased;
create view purchased as
select
    wl_product_id as pu_id,
    count(*) as pu_count
from
    web_pages,
    (select
        js.wl_user_id,
        js.wl_product_id,
```

```
        js.wl_webpage_name
     from web_logs
        lateral view
            json_tuple
            (web_logs.line,
            'wl_user_id', 'wl_product_id',
            'wl_webpage_name'
             ) js as
        wl_user_id, wl_product_id,
        wl_webpage_name
    where
        js.wl_user_id is not null
        and js.wl_product_id is not null
     ) logs
where
    wl_webpage_name = w_web_page_name
    and w_web_page_type = 'add to cart'
group by wl_product_id;

select
    i_product_id,
    (br_count-pu_count) as cnt
from
    browsed, purchased, product
where
    br_id = pu_id
    and br_id = i_product_id
order by cnt desc
limit 5;
```

- Q_7 :

```
drop view if exists sessions;

create view sessions as
select
    uid, item, wptype, tstamp,
    concat(sessionize.uid,
           concat('_',sum(new_session)
           over (partition by sessionize.uid
           order by sessionize.tstamp))
          ) as session_id
from (
 select
   logs.wl_user_id            as uid,
   logs.wl_item_id            as item,
   w.w_web_page_type          as wptype,
   unix_timestamp(logs.wl_ts) as tstamp,
   case
     when (unix_timestamp(logs.wl_ts)
           - lag(unix_timestamp(logs.wl_ts))
           over (partition by logs.wl_user_id
           order by logs.wl_ts)
          ) >= 600
     then 1
     else 0
   end as new_session
   from
      web_pages w,
      (select
          js.wl_user_id, js.wl_item_id,
          js.wl_webpage_name, js.wl_ts
       from web_logs
          lateral view
             json_tuple
             (web_logs.line,
             'wl_user_id', 'wl_item_id',
             'wl_webpage_name', 'wl_timestamp'
             ) js as
          wl_user_id, wl_item_id,
          wl_webpage_name, wl_ts
       where
          js.wl_user_id is not null
          and js.wl_item_id is not null
      ) logs
 where
    logs.wl_webpage_name = w.w_web_page_name
 cluster by uid
) sessionize
cluster by sessionid, uid, tstamp;

select
    c.c_user_id,
    c.c_name,
    count(*) as cnt_se
from
    sessions s,
    user c
where
    c.c_user_id = s.uid
group by c_user_id, c_name
having cnt_se > 10
order by cnt_se desc
limit 50;
```

- Q_9 :

```
drop view if exists sessions;

create view sessions as
select
    uid, tstamp,
    concat(sessionize.uid,
           concat('_', sum(new_session)
           over (partition by sessionize.uid
           order by sessionize.tstamp))
          ) as session_id
from (
 select
   logs.wl_user_id            as uid,
   unix_timestamp(logs.wl_ts) as tstamp,
   case
      when (unix_timestamp(logs.wl_ts)
            - lag (unix_timestamp(logs.wl_ts))
            over (partition by logs.wl_user_id
            order by logs.wl_ts)) >= 600
      then 1
      else 0
   end as new_session
   from
      web_logs
        lateral view
           json_tuple
           (web_logs.line, 'wl_user_id',
            'wl_item_id', 'wl_timestamp'
           ) logs as
        wl_user_id, wl_item_id, wl_ts
    where
        logs.wl_user_id is not null
    cluster by uid
) sessionize
cluster by sessionid, uid, tstamp;

select
    c_user_id,
    c_name,
    count(*)/24 as cnt
from
    sessions s,
    user c
where
    s.uid = c.c_user_id
group by c.c_user_id, c.c_name
order by cnt desc
limit 10;
```

- Q_{13} :

```
drop view if exists sessions;

create view sessions as
select
  uid,
  sessionid,
```

```
  min(tstamp) as startTime,
  max(tstamp) as endTime
from (
select
   uid, tstamp,
   concat(sessionize.uid,
          concat('_', sum(new_session)
          over (partition by sessionize.uid
          order by sessionize.tstamp))
         ) as session_id
from (
     select
       logs.wl_user_id             as uid,
       unix_timestamp(logs.wl_ts)  as tstamp,
       case
         when (unix_timestamp(logs.wl_ts)
               - lag (unix_timestamp(logs.wl_ts))
               over (partition by logs.wl_user_id
               order by logs.wl_ts)) >= 600
         then 1
         else 0
       end as new_session
     from (
       select
           js.wl_user_id, js.wl_item_id,
           js.wl_ts
       from web_logs
            lateral view
                json_tuple
                (web_logs.line, 'wl_user_id',
                'wl_item_id', 'wl_timestamp'
                ) js as
           wl_user_id, wl_item_id, wl_ts
       where
            js.wl_user_id is not null
            and js.wl_item_id is not null
        ) logs
     cluster by uid
    ) sessionize
  cluster by uid, session_id
) temp
group by uid, sessionid;

select
   avg(s.endTime-s.startTime)
from
   sessions s;
```

- Q_{14} :

```
select
    purchase_year,
    avg(items_per_user)
from
  (select
     userid as userid,
     year(to_date(dates[size_dates-1]))
      as purchase_year,
     sum(cart_items) as items_per_user
   from matchpath
   (on
     (select
          js.wl_user_id,
          js.wl_product_id,
          js.wl_webpage_name,
          js.wl_ts
       from web_logs
            lateral view
              json_tuple
              (web_logs.line,
                'wl_user_id', 'wl_product_id',
                'wl_webpage_name', 'wl_timestamp'
              ) js as
           wl_user_id, wl_product_id,
           wl_webpage_name, wl_ts
        where
           js.wl_user_id is not null
      ) n_logs
        partition by wl_user_id
        order by wl_ts
        arg1('A+.B'), arg2('A'),
        arg3(wl_webpage_name in ('webpage#01',
        'webpage#02','webpage#03','webpage#04',
        'webpage#05','webpage#06','webpage#07',
        'webpage#08','webpage#09','webpage#10',
        'webpage#11','webpage#12','webpage#13',
        'webpage#14','webpage#15','webpage#16',
        'webpage#17','webpage#18','webpage#19',
        'webpage#20')),
        arg4('B'),
        arg5(wl_webpage_name in('webpage#21',
        'webpage#22','webpage#23','webpage#24',
        'webpage#25')),
        arg6('tpath[0].wl_user_id as userid,
        (size(tpath.wl_product_id)-1)
        as cart_items,
        tpath.wl_ts as dates,
        size(tpath.wl_ts) as size_dates')
    ) group by userid,
               cart_items,
               dates[size_dates-1]
) as t
group by purchase_year
order by purchase_year;
```

- Q_{16} :

```
select
wl_webpage_name,
count(*) as cnt
from
web_logs
 lateral view
  json_tuple
      (web_logs.line,
   wl_webpage_name
  ) logs as wl_webpage_name
where
wl_webpage_name is not null
group by wl_webpage_name
order by cnt desc
limit 10;
```

- Q_{17} :

```
select
    wl_webpage_name,
    count(*) as cnt
from
    web_logs
      lateral view
       json_tuple
       (web_logs.line,
        'wl_webpage_name', 'wl_timestamp'
       ) logs as wl_webpage_name, wl_timestamp
where
    wl_webpage_name is not null
and
    to_date(wl_timestamp) >= '2013-02-14'
    and to_date(wl_timestamp) < '2014-02-15'
group by wl_webpage_name
order by cnt desc
limit 10;
```

- Q_{19} :

```
select
    day(to_date(wl_timestamp))   as d,
    month(to_date(wl_timestamp)) as m,
    year(to_date(wl_timestamp))  as y,
    count(*) as PageViews
from
    web_logs
      lateral view
       json_tuple
```

```
       (web_logs.line,
         'wl_timestamp'
       ) logs as wl_timestamp
group by wl_timestamp
order by PageViews desc
limit 10;
```

- Q_{20} :

```
drop view if exists temp1;

create view temp1 as
select
    c.c_user_id as o_user,
    sum(ws.ws_quantity * i.i_price)
    as online_revenue
from
    web_sales ws,
    user c,
    product i
where ws.ws_user_id = c.c_user_id
     and ws.ws_user_id is not null
     and ws.ws_product_id = i.i_product_id
group by c.c_user_id
order by c.c_user_id asc;

drop view if exists temp2;

create view temp2 as
select
    c.c_user_id as i_user,
    sum(ss.ss_quantity * i.i_price)
    as instore_revenue
from
    store_sales ss,
    user c,
    product i
where ss.ss_user_id = c.c_user_id
    and ss.ss_user_id is not null
    and ss.ss_product_id = i.i_product_id
group by c.c_user_id
order by c.c_user_id asc;

drop table if exists q20_results;

create table q20_results (
  online_segment      bigint,
  instore_segment     bigint
)
row format
delimited fields terminated by ','
lines terminated by '\n'
stored as textfile;

insert into table q20_results
select
  sum(case
      when t1.online_revenue
        >= t2.instore_revenue
      then 1
      else 0 end) as online_revenue,
  sum(case
      when t1.online_revenue
         < t2.instore_revenue
      then 1
      else 0 end) as instore_revenue
from user c join temp1 t1
on c.c_user_id = t1.o_user
join temp2 t2
on c.c_user_id = t2.i_user;
```

- Q_{21} :

```
select
    path_to_purchase,
    count(*) as freq
from matchpath
(on
  (select
      js.wl_user_id,
      js.wl_product_id,
      js.wl_webpage_name,
      js.wl_ts
   from web_logs
      lateral view
        json_tuple
        (web_logs.line,
          'wl_user_id', 'wl_product_id',
          'wl_webpage_name', 'wl_timestamp'
        ) js as
        wl_user_id, wl_product_id,
        wl_webpage_name, wl_ts
   where
      js.wl_user_id is not null
) n_logs
    partition by wl_user_id
    order by wl_ts
    arg1('other+.purchase'),
    arg2('other'),
    arg3(wl_webpage_name not in ('webpage#21',
    'webpage#22','webpage#23','webpage#24',
    'webpage#25')),
    arg4('purchase'),
    arg5(wl_webpage_name in ('webpage#21',
    'webpage#22','webpage#23','webpage#24',
    'webpage#25')),
    arg6('tpath.wl_webpage_name
         as path_to_purchase')
)
group by  path_to_purchase
order by  freq desc
limit 5;
```

- Q_{22} :

```
select
   day(to_date(wl_timestamp))   as d,
   month(to_date(wl_timestamp)) as m,
   year(to_date(wl_timestamp))  as y,
   count(distinct wl_user_id)   as uniqueVisitors
from
   web_logs
     lateral view
       json_tuple
       (web_logs.line,
        'wl_user_id', 'wl_timestamp'
       ) l as wl_user_id, wl_timestamp
where
   wl_user_id is not null
group by wl_timestamp
order by uniqueVisitors desc
limit 10;
```

- Q_{23} :

```
select
   c_user_id,
   c_name,
   count(*) as visits
from
   (select
     lg.wl_user_id
    from web_logs wl
      lateral view
       json_tuple
       (wl.line, 'wl_user_id'
       ) lg as wl_user_id
    where
     lg.wl_user_id is not null
    ) l,
   user
where
   l.wl_user_id = c_user_id
group by c_user_id, c_name
order by visits desc
limit 10;
```