CoreBigBench: Benchmarking Big Data Core Operations

Todor Ivanov
Frankfurt Big Data Lab, Goethe University
Frankfurt am Main, Hessen, Germany
todor@dbis.cs.uni-frankfurt.de

Ahmad Ghazal
Facebook Corporation
Seattle, WA, USA
ahmadghazal64@fb.com

Alain Crolotte
Teradata Corporation
El Segundo, CA, USA
alain.crolotte@teradata.com

Pekka Kostamaa
Teradata Corporation
El Segundo, CA, USA
pekka.kostamaa@teradata.com

Yoseph Ghazal
University of California, Irvine
Irvine, CA, USA
yghazal@uci.edi

ABSTRACT

Significant effort was put into big data benchmarking with focus on end-to-end applications. While covering basic functionalities implicitly, the details of the individual contributions to the overall performance are hidden. As a result, end-to-end benchmarks could be biased toward certain basic functions. Micro-benchmarks are more explicit at covering basic functionalities but they are usually targeted at some highly specialized functions. In this paper we present CoreBigBench, a benchmark that focuses on the most common big data engines/platforms functionalities like scans, two way joins, common UDF execution and more. These common functionalities are benchmarked over relational and key-value data models which covers majority of data models. The benchmark consists of 22 queries applied to sales data and key-value web logs covering the basic functionalities. We ran CoreBigBench on Hive as a proof of concept and verified that the benchmark is easy to deploy and collected performance data. Finally, we believe that CoreBigBench is a good fit for commercial big data engines performance testing focused on basic engine functionalities not covered in end-to-end benchmarks.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**.

KEYWORDS

Big Data Benchmarking, Benchmark Operations, BigBench.

ACM Reference Format:
Todor Ivanov, Ahmad Ghazal, Alain Crolotte, Pekka Kostamaa, and Yoseph Ghazal. 2020. CoreBigBench: Benchmarking Big Data Core Operations. In *Workshop on Testing Database Systems (DBTest'20), June 19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3395032.3395324

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DBTest'20, June 19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8001-0/20/06…$15.00
https://doi.org/10.1145/3395032.3395324

1 INTRODUCTION

Big data systems continue to get attention in the industry and academia. Those big data systems made it necessary to develop benchmarks that assess their functionality and performance. These benchmarks can be broken down into two types: benchmarks simulating a real world application and micro-benchmarks that focus on specific functionality. The application benchmark starts with a business problem and proposes a data model and workload specifications as the components of the benchmark. Examples of application based benchmarks for big data are : BigBench [9, 11, 16, 19, 25, 28], BigBench V2 [15] and others [1].

We believe that application driven benchmarks are important since they mimic real life scenarios. However, they could be biased to certain functionalities of the system since the workload is the main driver for covering those functionalities. Another issue of application driven benchmarks is that they are subject to special purpose tuning that does not provide a fair comparison among different systems. In this paper, we address the limitations of application based benchmarks by proposing a benchmark that covers all or the most common basic functionalities of big data systems. The proposed benchmark is complimentary to application level benchmarks and not intended to be a replacement for them. Our proposal is also different from micro-benchmarks that focus on a single operation like WordCount [6], Pi [8], Terasort [7] or a single system functionality like TestDFSIO [2].

In this paper we propose CoreBigBench, a benchmark that focuses on assessing the performance of core (or basic) operations of big data engines. Our focus is on system operations that are common across big data systems and avoided system specific ones. The most common core operations include scans, two way joins, aggregations and window functions. In addition, two fundamental functions: sessionize and path are included as basic operations given their popularity in big data processing (see [9, 11, 16, 19, 25, 28], BigBench V2 [15]). CoreBigBench includes these basic operations on relational and key-value data models since these two are the most common data sources in big data. The key-value data model also indirectly covers a basic and fundamental late binding processing as explained in [15].

As any benchmark, the main components of CoreBigBench is the data model and workload. The data model leverages the data model of BigBench V2 since it already have a well founded relational data (sales information) and key-value semi-structured data (product

reviews). The workload is composed of 22 queries that cover the basic operations applied on both the relational sales data and key-value web logs.

We implemented the CoreBigBench queries in HiveQL and executed them on a 4 node cluster running Hive. The proof of concept implementation and execution showed the feasibility of the benchmark in terms of setup and deployment.

The rest of the paper is organized as follows. Related work is shown in section 2 and section 3 covers the details of the benchmark data model and workload. The proof of concept implementation in Hive is presented in section 4. Finally, we conclude the paper through section 5.

2 RELATED WORK

There are multiple existing micro-benchmarks that focus on testing single operations like WordCount [6], Pi [8], Terasort [7] or a single system functionality like TestDFSIO [2]. Initially these micro-benchmarks were targeting the MapReduce framework as part of Hadoop, but later they evolved to specifically test other tools from the big data ecosystem.

For example, HiveBench [23] is a micro-benchmark that stresses the main Hive functionalities (aggregation and join) and was included in the HiBench [20] micro-benchmark suite. It was later extended with one external script query and implemented on multiple other engines (RedShift, Tez, Shark and Impala) by the AMP Lab at Berkeley [4]. However, the set of implemented queries is very limited (only 4) and does not really cover all basic operations and data models. HiveRunner [3] is a unit test framework used in the Hive development. SparkBench [22] is micro-benchmark designed for testing various Spark operations, whereas Spark-sql-perf [13] is focused on the SparkSQL features. PigMix [5] focuses on the Pig features. BigFUN [24] focuses on stressing the AsterixDB features. Tempto [26] and Benchto [10] are particularly developed to test Presto.

As mentioned above, almost every SQL-on-Hadoop engine like Hive, Spark SQL, Presto, etc. comes with its own specific micro-benchmark to stress test its features. The challenge is that each of these benchmarks implement a different set of core operations that makes comparison between the engines impossible. At the same time there is no standardized set or specification of core operations for big data analytical engines that can be used as a reference when evaluating the engines.

Core DBMS operations benchmark was proposed in [12] that covered scans, aggregations, joins and other core relational operators. In this work, we make a similar attempt to cover core operations for big data engines.

3 BENCHMARK SPECIFICATIONS

This section details the specification of CoreBigBench which include the data model in section 3.1 and the 22 workload queries in section 3.2. We choose the geometric mean of the execution time of the workload queries as a metric for the benchmark.

3.1 Data Model

We leverage the data model and the scalable data generator of BigBench V2 [15] which provide us with all the data needed for our purpose given that it has all the structured, semi-structured and unstructured data. Also, the BigBench V2 [15] data model is a major improvement over the original data model in BigBench [16]. However, we re-iterate the high level aspects of the model to help with the description of workload and proof of concept in the subsequent sections.

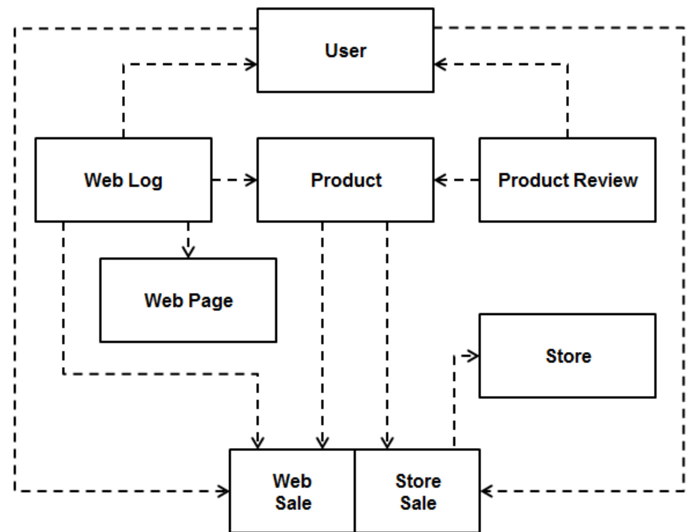

Figure 1: CoreBigBench Data Model

Figure 1 depicts the data model we are using with six relational tables that represent the structured part of a retailer that sells products online and in brick and mortar stores. Users can browse web pages of the retailer and can buy products online. Their web access is captured for analysis and is stored in a semi-structured key-value JSON file called "Web Log" shown in Figure 1. Users can also write reviews about products offered by the retailer and these reviews are maintained as unstructured text in "Product Review".

BigBench V2 also has a custom data generator that scales linearly (web and store) sales, web logs and reviews. Users and products scale sub-linearly and web pages are assumed to be static. The proof of concept in section 4 provides more information on the data definition language (DDL) of the data sources and how they are loaded.

3.2 Workload

This section describes our 22 workload queries. Big data analytic involve a variety of basic relational operations like scans, binary joins, aggregations, data shuffle, etc.. Also, we treat some very common UDFs like path analysis (like click sequence) and sessionization (find users sessions) as core operations in our benchmark. Relational operators and UDFs can be executed on both structured and semi-structured data and we cover both variations as well. We also cover unstructured data through one common UDF that does sentiment analysis.

The first set of queries covers different variations of scans on structured data with different selectivity's and scans on the semi-structured data.

- Q_1: List all store sold products (items) together with their quantity. This query does a full table scan of the store data.
- Q_2: List all products (items) sold together in stores with their quantity sold between 2013-04-21 and 2013-07-03. This query tests scans with low selectivity 10% filter.
- Q_3 : List all products (items) together with their quantity sold between 2013-01-21 and 2014-11-10. Similar to Q_2 but with high selectivity (90%).
- Q_4: List names of all visited web pages. This query tests parsing the semi-structured web logs and scanning the parsed results. The query requires only one key from the web logs.
- Q_5: Similar to Q_4 above but returning a bigger set of keys. This variation measures the ability of the underlying system for producing a bigger schema out of the web logs.

The second set of queries below measures the cost of aggregations with variations on source of data and number of aggregate expressions.

- Q_6: Find total number of all stores sales. This query covers basic aggregations with no grouping. The query involves scanning store sales and to get the net cost of aggregations we deduct the cost of Q_1 from this query run time.
- Q_7: Find total number of visited web pages. This query requires parsing and scanning the web logs and therefore it is adjusted by subtracting Q_4 from its run time.
- Q_8: Find total number of store sales per product (item). This query is adjusted similar to Q_6.
- Q_9: Find number of clicks per product (item). This query also requires parsing the web logs and can be adjusted similar to Q_7.
- Q_{10}: Find a list of aggregations from store sales by customer. Aggregations include number of transactions, maximum and minimum quantities purchased in an order. This query also finds correlations between stores and products (items) purchased by a a customer. The purpose of this query is to test cases of a big set of aggregations.
- Q_{11}: This query has a simple objective like Q_{10} but applied to web logs. Again, the query need to be adjusted by removing the parsing and scan cost represented by Q_4.

Note that in a cluster system data need to be shuffled on the group key to do aggregation. In the structured case, we took out this factor by running the queries with group by on store sales versions partitioned on the group key. For example, Q_8 is ran on store sales partitioned on item ID and another version of store sales partitioned by customer ID is used for Q_{10}. We could not do the same for the aggregation queries on web logs since it is difficult to partition JSON file on a certain key.

We consider the shuffle by itself is a core operation in a big data system running on a cluster. To get that cost, we use another query Q_{12} which is the same as Q_8 but on store sales partitioned by customer (different than the group key). The shuffle cost is computed as run-time of Q_{12} minus run-time of Q_8.

Window functions are also common in big data processing and the next set of queries covers those core operations. The variations include source of data and use or no use of partitioning. No partitioning is a simple version of window function where the window is applied to the whole input data set. Using partitioning requires similar logic to group by in the aggregate case and we thought it is better to separate these two cases. All the queries using store sales are adjusted by the scan cost in Q1 and similarly queries on web logs are adjusted using Q_4.

- Q_{13}: Find row numbers of store sales records order by store id.
- Q_{14}: Find row numbers of web log records ordered by timestamp of clicks.
- Q_{15}: Find row numbers of store sales records order by store id for each customer. This query is similar to Q_{13} but computes the row numbers for each customer individually.
- Q_{16}: Same as Q_{14} where row numbers are computed per customer.

Binary joins are one of the core operations we consider. In a distributed system joining two tables can be done with data shuffle if the tables are co-located (partitioned on the join fields). If one or both tables are not partitioned on the join column then one or both tables need to be shuffled to perform the join. Q_{17} below is used to measure the performance of co-located joins and Q_{18} is used for non co-located joins.

- Q_{17}: Find all store sales with products that were reviewed. This query is a join between the stores sales and product reviews both partitioned on item ID.
- Q_{18}: Same as Q_{17} with different partitioning. (Table store sales is partitioned on customer ID and no partitioning on table product reviews.)

Big data processing use common functions for analtics and we consider some of those as core operations as well. The most common UDFs used in BigBench and BigBench V2 are: sessionize, path, sentiment analysis and Kmeans. The four queries below cover the UDF core operations.

- Q_{19}: List all customers that spend more than 10 minutes on the retailer web site. This query involves finding all sessions of all users and filtering them to those which are 10 minutes of less.
- Q_{20}: Find the 5 most popular web page paths that lead to a purchase. This query is based on finding paths in clicks that lead to purchases, aggregating the results and finding the top 5.
- Q_{21}: For all products, extract sentences from its product reviews that contain Positive or Negative sentiment and display the sentiment polarity of the extracted sentences.
- Q_{22}: Cluster customers into book buddies/club groups based on their in-store book purchasing histories. After model of separation is build, report for the analysed customers to which "group" they where assigned.

4 PROOF OF CONCEPT

The objective of our proof of concept is to show the feasibility of CoreBigBench rather than assessing the performance of one system over the other. On that basis we ran CoreBigBench on a small cluster with 4 nodes with Hadoop (Cloudera CDH 5.16.2) and Hive 1.10.

4.1 Data Generation and Loading

We used the BigBench V2 data generator [15, 27] to generate data with scale factor 10. Then using a HiveQL script, we created the data model schema and loaded the six structured tables, the product reviews and the external web logs table in Hive. The HiveQL definition for the table *store sales* 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 store_sales;
CREATE TABLE store_sales
( ss_transaction_id      bigint,
  ss_customer_id         bigint,
  ss_store_id            bigint,
  ss_item_id             bigint,
  ss_quantity            int,
  ss_ts                  string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
STORED AS TEXTFILE LOCATION 'hdfsDataPath/user';
```

In some of the queries like Q_8 and Q_{10} we use variations of the *store sales* table partitioned by item or by customer ID. This can be easily done by extending the table schema with the *PARTITIONED by (key)* clause.

The web logs represent the semi-structured data as JSON format records and are stored in a file called *clicks.json* on HDFS. 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';
```

It is important to mention that the web logs (around 40GB) represent the biggest chunk of data compared to the structured tables (around 51MB). The data loading times per table are also provided in Table 1.

Table 1: Data size and loading time

	Scale Factor 1		Scale Factor 10	
Table Name	**Data Size**	**Loading (sec.)**	**Data Size**	**Loading (sec.)**
user	256 KB	16.63	456 KB	19.19
product	39 KB	13.16	46 KB	17.33
product review	6.9 MB	15.20	13 MB	13.40
web log	22 GB	0.02	40 GB	0.04
web page	687 B	13.05	687 B	14.80
web sale	9.7 MB	16.10	18 MB	14.63
store sale	10 MB	15.03	19 MB	15.20
store	6.6 KB	13.88	6.2 KB	13.67
Total:	2̃2.03 GB	103.08	40.05 GB	108.2

4.2 Query Implementation

This section describes our HiveQL implementation of the queries, defined in section 3.2 and available in github [21]. Similar to BigBench [19] and BigBench V2 [27], we chose Hive [17] for our reference implementation as the most commonly used data warehouse engine on Hadoop. However, our goal is to keep the CoreBigBench query definitions independent of particular engine implementation and allow syntax and engine optimizations. We divide the 22 queries into three groups: queries on structured data, queries on semi-structured data and UDF functions. Due to space limitations next we look at the query implementation of only a subset of the queries.

For example, Q_2 is implementing scan over the store sales table with additional filter on timestamp that selects around 10% of all rows. The HiveQL code is below:

```
SELECT ss_item_id, ss_quantity FROM store_sales
WHERE to_date(ss_ts) >= '2013-04-21'
AND to_date(ss_ts) < '2013-07-03';
```

The set of queries accessing the key-value web logs in the JSON file need to implement late binding, so that the keys are parsed at runtime in the same way as in BigBench V2 [15, 27]. 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* [18] and extracts keys from the JSON records. For example, Q_4, defined in section 3.2, uses the *json_tuple* UDTF to extract only the *wl_webpage_name* key from each JSON record:

```
SELECT wl_webpage_name
FROM web_logs
    lateral view json_tuple(
    web_logs.line,'wl_webpage_name'
    )logs as wl_webpage_name
WHERE wl_webpage_name IS NULL;
```

In section 3.2 we identified multiple common UDFs (sessionize, path, sentiment analysis and Kmeans) that can be applied both on the structured tables and the semi-structured key-value web log data. For example, we adapt one of the BigBench queries that uses the Kmeans algorithm as Q_{22}. In the preparation phase of the query, the filtered product (item) IDs are stored in a temporary table called *q22_prep_data*. However, instead of using Mahout or Spark MLlib in the implementation, we used the Facebook Hive UDF library [14], which allowed us to call the KMeans UDF directly in HiveQL without external library calls:

```
set cluster_centers=8;
set clustering_iterations=20;

SELECT kmeans(
    collect_list(array(id1, id3, id5, id7, id9,
    id11, id13, id15, id2, id4, id6, id8, id10,
    id14, id16)),
    ${hiveconf:cluster_centers},
    ${hiveconf:clustering_iterations}) AS out
FROM q22_prep_data;
```

4.3 Experimental Results

We executed all queries implemented in HiveQL [21] on our cluster processing data generated with scale factor 10. Again, we divided the results in three groups: relational, key-value and UDFs.

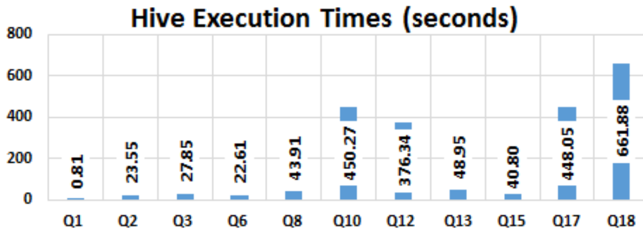

Figure 2: Queries on Structured Data

Figure 2 depicts the execution times of all queries on the structured data. We calculate the execution times for the different queries as specified in the workload section 3.2. The geometric mean of all query times in this group is 62.07 seconds.

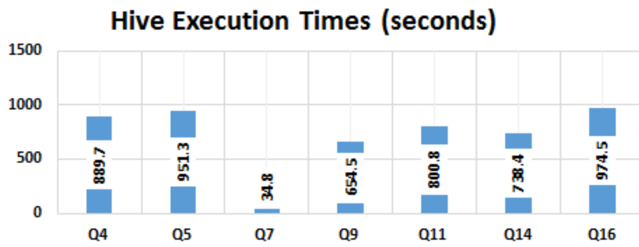

Figure 3: Queries on Semi-structured Data

Figure 3 depicts the execution times of the group of queries operating on the semi-structured key-value data. Q_4 performs a simple scan operation that involves parsing all the JSON records on the fly and extracting only the necessary attributes. Since this is a core key-value operation, we deduct this time from all the other group queries. The geometric mean of all query times in this group is 525.88 seconds.

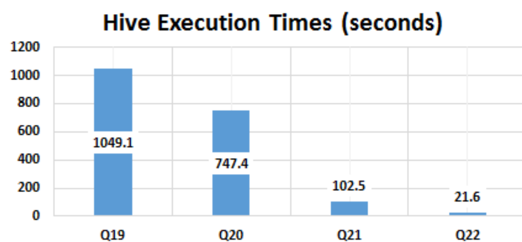

Figure 4: UDF Functions

Figure 4 depicts the execution times of the group of UDF functions (sessionize, path, sentiment analysis and Kmeans). Two of them (Q_{19} and Q_{20}) operate on the semi-structured key-value data, whereas the other two (Q_{21} and Q_{22}) operate on the structured and unstructured data. Therefore, we deduct the basic key-value scan (Q_4) operation time from Q_{19} and Q_{20} and respectively the simple table scan (Q_1) time from Q_{21} and Q_{22}.

5 CONCLUSIONS

In this paper we presented CoreBigBench, a benchmark that focuses on big data engines core operations like scans, two way joins, UDF functions and more. The underlying data model was inspired by BigBench V2 and utilizes the structured relational sales data, the key-value web logs data and unstructured product reviews. The benchmark consists of 22 queries applied on sales data and key-value web logs that cover the basic functionalities. We implement CoreBigBench in Hive as a proof of concept and verified that the benchmark is easy to deploy and stresses the desired system features. CoreBigBench also can be used for regression testing of commercial big data engines and can be used complimentary to an end-to-end benchmarks like BigBench.

In the future we plan to extend the benchmark with more queries that address data extraction, loading, transformation and cleaning operations as well as data compression and decompression.

ACKNOWLEDGMENTS

This work has been partially funded by the European Commission H2020 project DataBench - Evidence Based Big Data Benchmarking to Improve Business Performance, under project No. 780966. This work expresses the opinions of the authors and not necessarily those of the European Commission. The European Commission is not liable for any use that may be made of the information contained in this work. The authors thank all the participants in the project for discussions and common work.

REFERENCES

[1] 2016. Big Data Benchmarks, Performance Optimization, and Emerging Hardware *(Lecture Notes in Computer Science)*, Jianfeng Zhan, Rui Han, and Roberto V. Zicari (Eds.), Vol. 9495. Springer.
[2] 2018. Apache Hadoop DFSIO benchmark. http://svn.apache.org/repos/asf/hadoop/common/tags/release-0.13.0/src/test/org/apache/hadoop/fs/TestDFSIO.java
[3] Klarna AB. 2020. HiveRunner. https://github.com/klarna/HiveRunner
[4] AMP Lab. 2013. AMP Lab Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/
[5] Apache. 2013. PigMix. https://cwiki.apache.org/confluence/display/PIG/PigMix
[6] Apache. 2020. WordCount. https://cwiki.apache.org/confluence/display/HADOOP2/WordCount
[7] Apache Hadoop. 2015. TeraSort. http://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html
[8] Apache Software Foundation. 2015. Package hadoop.examples.pi. http://hadoop.apache.org/docs/r0.23.11/api/org/apache/hadoop/examples/pi/package-summary.html
[9] Chaitanya K. Baru, Milind A. Bhandarkar, Carlo Curino, Manuel Danisch, Michael Frank, Bhaskar Gowda, Hans-Arno Jacobsen, Huang Jie, Dileep Kumar, Raghunath Othayoth Nambiar, Meikel Poess, Francois Raab, Tilmann Rabl, Nishkam Ravi, Kai Sachs, Saptak Sen, Lan Yi, and Choonhan Youn. 2014. In *Proc. of the 6th TPCTC 2014, Hangzhou, China, Sept. 1-5, 2014*. 44–63.
[10] Benchto. 2020. Benchto. https://github.com/prestodb/benchto
[11] Badrul Chowdhury, Tilmann Rabl, Pooya Saadatpanah, Jiang Du, and Hans-Arno Jacobsen. 2013. A BigBench Implementation in the Hadoop Ecosystem. In *Proc. of the 2013 WBDB.cn, Xi'an, China, July 16-17, 2013 and WBDB.us, San José, CA, USA, October 9-10, 2013*. 3–18.
[12] Alain Crolotte and Ahmad Ghazal. 2010. Benchmarking Using Basic DBMS Operations. In *PRoc of the 2nd TPCTC 2010, Singapore, September 13-17, 2010*. 204–215.

[13] Databricks. 2020. Spark-SQL-perf. https://github.com/databricks/spark-sql-perf
[14] facebook-hive-udfs. 2019. https://github.com/brndnmtthws/facebook-hive-udfs/blob/master/src/main/java/com/facebook/hive/udf/UDFKmeans.java.
[15] Ahmad Ghazal, Todor Ivanov, Pekka Kostamaa, Alain Crolotte, Ryan Voong, Mohammed Al-Kateb, Waleed Ghazal, and Roberto V. Zicari. 2017. BigBench V2: The New and Improved BigBench. In *ICDE 2017, San Diego, CA, USA, April 19-22*. 1225–1236.
[16] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. 2013. BigBench: towards an industry standard benchmark for big data analytics. In *Proc. of the ACM SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 1197–1208.
[17] Hive. [n.d.]. hive.apache.org.
[18] Hive Lateral View. 2013. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+LateralView.
[19] Intel. 2018. https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench
[20] Intel. 2020. HiBench Suite. https://github.com/intel-hadoop/HiBench
[21] Todor Ivanov. 2020. CoreBigBench. https://github.com/t-ivanov/CoreBigBench
[22] Min Li. 2015. SparkBench. https://bitbucket.org/lm0926/sparkbench
[23] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proc. of the ACM SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. ACM, 165–178.
[24] Pouria Pirzadeh. 2015. BigFUN. https://github.com/pouriapirz/bigFUN
[25] Tilmann Rabl, Ahmad Ghazal, Minqing Hu, Alain Crolotte, Francois Raab, Meikel Poess, and Hans-Arno Jacobsen. 2012. BigBench Specification V0.1 - BigBench: An Industry Standard Benchmark for Big Data Analytics. In *Proc. of the 1st WBDB 2012, San Jose, CA, USA, May 8-9, 2012, and 2nd WBDB 2012, Pune, India, December 17-18, 2012*. 164–201.
[26] Tempto. 2020. Tempto. https://github.com/prestodb/tempto
[27] Todor Ivanov. 2018. BigBench V2. https://github.com/t-ivanov/BigBenchV2
[28] TPC. 2018. http://www.tpc.org/tpc_documents_current_versions/pdf/tpcx-bb_v1.2.0.pdf