A Performance Study of Big Data Analytics Platforms

Pouria Pirzadeh
Microsoft [§]
United States
pouriap@microsoft.com

Michael Carey
University of California, Irvine
United States
mjcarey@ics.uci.edu

Till Westmann
Couchbase, Inc.
United States
till@couchbase.com

Abstract*—Big Data analytics has become an invaluable tool in a wide variety of businesses for exploiting the wealth of Big Data that they now have access to. As a result, various solutions within different categories of Big Data systems are emerging to meet their needs. In this paper we use the TPC-H benchmark to compare the performance of four Big Data systems picked from the major categories of Big Data platforms: a commercial parallel relational database (from the traditional DBMS world), Hive and Spark SQL (from the SQL-on-Hadoop world), and AsterixDB (from the world of NoSQL systems). All of these systems have sufficiently rich query APIs and runtime systems to run TPC-H in its full form. On the other hand, the systems also have major differences in terms of their architectures, preferred storage formats, support for complex schema definitions, and approaches to query processing. This makes them a very interesting set of representative Big Data systems to compare. We present the results that we obtained through running these systems at different TPC-H scales using various settings, and we analyze a selected set of interesting query results in more detail to explore the trade-offs between performance, storage formats, and schema definitions. A follow-up discussion is included as well to summarize the lessons learned from this effort.

***Keywords*-Big Data; Performance Evaluation; Benchmarking.**

I. Introduction

Big Data is turning out to be an essential factor in decision making processes at leading companies that seek to outperform their competitors. Big Data analytics is an important tool that an organization can exploit to use the wealth of data it has access to. Big Data analytics is about collecting, storing, and analyzing large volumes of data efficiently with the goal of extracting invaluable, but often hidden, information from it. It is expected that Big Data analytics will redefine the competitive landscape of various industries in the near future, and companies that do not adopt a Big Data analytics strategy risk losing market share and momentum [1].

Unlike transaction processing (OLTP-style) workloads, which are characterized by a large number of short and mostly concurrent requests initiated by many users, Big Data analytics workloads are characterized by fewer requests, mostly submitted by experts and business analysts, with each query tending to be complex and resource intensive. An analytical workload normally has a set of characteristics that includes: extreme data volume, complex data model, bulk operations, multi-step analysis algorithms, and intermediate staging of temporary data [2]. The response times in an analytical workload mostly tend to be on the order of tens to hundreds of seconds and depending on the available resources, CPU, I/O, the network, or a combination thereof could be the processing bottleneck.

Because of the inherent differences between OLTP and OLAP workloads, the serving systems for them are built in different manners. Key design points in an analytical processing framework include functional richness and efficient resource usage. Moreover, the requirements of applications using analytical frameworks have evolved over time and become more complex. Therefore, the richness of the API(s) to interact with such systems and adequate expressive power in their supported query languages are among other major design points.

Parallel relational databases and data warehouses were the prominent solutions for serving analytical workloads for a long time. With the advent of Big Data, however, various new frameworks have been developed to manage and run analytics to serve different applications such as log analysis, text analytics, or the task of organizational decision-making. Examples of these systems include Hive [3], Presto [4], Impala [5], Dremel [6], Drill [7], Spark SQL [8], and IBM Big SQL [9]. These frameworks normally require an environment with characteristics such as high storage capacity, fast data transfer capacity, and large volumes of available memory in order to achieve a desired performance level and proper horizontal scaling. In terms of their data storage, all of these systems can operate on data which is stored in HDFS. In terms of their APIs, they all provide SQL-based languages to describe queries. However, they differ significantly in terms of their architecture, optimization and code generation techniques, and run-time processing approaches. As analytical workloads tend to be complex and resource-intensive, these differences can considerably impact their performance.

In this paper, we use the TPC-H benchmark to evaluate four Big Data systems selected from different regions of the Big Data platform design space. We chose TPC-H because its data model and query workload are complex enough to serve as a reasonable set of analytics tasks. Moreover, many vendors are still using this benchmark to evaluate and report on the performance of their technology. The systems we are comparing to each other here are Apache Hive and Spark SQL (chosen from the SQL-on-Hadoop world), System-X, which is a commercial

[§]Work done while at the University of California, Irvine.

parallel relational DBMS, and Apache AsterixDB [10] (chosen from the world of NoSQL systems). All four have sufficiently rich SQL-based query APIs and underlying runtime systems to run all queries in the TPC-H query suite. On the other hand, their architectures, preferred storage formats, support for auxiliary index structures, and level of richness and flexibility in schema definition have major differences that makes them an interesting set of systems to compare. The main goal in this paper is to study the performance of these systems and their scale-up behavior with a focus on the impact of different storage formats, ability to support complex schemas (e.g., nesting in the definition and storing of data), and different query optimization techniques. In the remainder of this paper we discuss some related work and review the important details about the systems and workload, we consider. The full set of results and details regarding our experiments come next, followed by a deeper analysis of a number of interesting queries. We conclude the paper with a discussion on the lessons we learned from this effort.

II. Related Work

The performance evaluation of data stores for analytical workloads has long been a major topic in the context of Big Data benchmarking. Among the released benchmarks by TPC, TPC-H [11] and TPC-DS [12] include analytical and OLAP-style queries with different levels of complexity. Both benchmarks have been used in various works on Big Data benchmarking. From the Big Data community, one of the very first works on the topic of Big Data analytics was [13], which compared MapReduce (Hadoop) to RDBMSs. Being focused on basic architectural differences of the frameworks such as their programming model, expressiveness, and fault-tolerance, the analytical workload in [13] included four simple non-TPC queries: filtered selection, grouped aggregation, join, and UDF aggregation. A more recent study that compared Hive against SQL Server parallel data warehouse (PDW) using the TPC-H benchmark was [14]. That work showed that a relational system can provide a significant performance advantage over Hive. Another recent paper [15] compared the performance of Hive and Impala using TPC-H and a TPC-DS inspired workload. With a focus on the I/O efficiency of columnar storage formats, the results in [15] showed that Impala was faster than Hive for different queries. Its main conclusion was to reaffirm the clear advantages of using a shared-nothing dataflow architecture for analytical SQL queries over a MapReduce-based runtime. Our work in this paper has some overlap with [14] and [15] in terms of workload (using TPC-H) and one of the evaluated frameworks (Hive). However, beside the fact that the set of systems and settings used for the experiments are different here, we also focus on how support for non-1NF schemas and nested data types along with differences in query optimization affect the overall performance of an analytical workload.

There have also been recent Big Data benchmarks that consider a schema with nesting for semi-structured and unstructured data in their workloads (such as [16]). In the context of TPC-H, the Impala developers have modified the benchmark's schema, added nesting to it, and used it to evaluate the support of complex types in their system [17]. Their approach has some differences with ours and in our view did not follow the most sensible way of denormalizing (nesting) a relational schema considering the entities and their relationships of the TPC-H data.

III. Systems overview

This section provides an overview of the systems and storage formats considered for the purpose of our evaluation. It also goes over the important details of TPC-H and describes the changes that we made to its original schema to include nesting in the data and query set for our experiments.

A. Systems Overview

1) Apache Hive: Hive [3] is a data warehouse on top of Hadoop [18]. It provides a SQL-like interface through its query language, HiveQL, on tables created on existing data files in HDFS. Hive supports various file formats for the data which have significant performance differences such as sequence and text files and ORC [19]. A client's query in Hive gets compiled and turned to an optimized DAG of map and reduce jobs to be executed by Hadoop. Apache Tez [20] is a newer runtime engine that can also be used with Hive; Tez is an open-source framework designed to build dataflow-driven processing runtimes [21]. Tez generalizes the MapReduce paradigm to execute complex computations more efficiently through parallel runtime producer and consumer tasks. Hive's performance improves with Tez, as Tez uses pipelining and in-memory writes and it allows for multiple reduce stages.

2) Apache AsterixDB: AsterixDB [10] is an open source Big Data management system (BDMS) with a rich set of features for storing, managing and analyzing semi-structured data. Its SQL-based query language is SQL++ and its data model (ADM) is a "super-set" of JSON. AsterixDB has built-in support for collection data types (arrays and bags) and supports data types with an arbitrary level of nesting. AsterixDB uses a hash-partitioned LSM-based storage layer as its native storage [22] while it also has support for HDFS as external storage. AsterixDB provides users with different types of indexing structures such as B+ Trees, spatial indices (R-Tree), and text indices (inverted index). As the execution engine, AsterixDB uses Hyracks [23], a data-parallel runtime platform for shared-nothing clusters. Clients can use the available HTTP API in AsterixDB to submit queries. The queries are first compiled into an algebraic form that is then optimized by a rule-based optimizer and ultimately turned into corresponding Hyracks jobs. Hyracks executes the jobs and the results are delivered back to the client synchronously or can be fetched asynchronously.

3) System-X: System-X is a commercial, parallel, shared-nothing, relational database system (unnamed for licensing reasons). It uses the relational data model to define schemas and uses SQL as the query language. System-X partitions data horizontally among the cluster nodes. The data is stored in tables and the storage is managed using efficient native RDBMS storage technology. System-X has support for different types of indices and integrity constraints. A client can submit a query through any of system's supported APIs, such as standard, vendor-provided JDBC drivers. Having a mature cost-based query optimizer (which can be equipped with statistics about the data), an input request gets converted into an optimized query plan whose execution is then done in parallel by the nodes in the cluster.

4) Apache Spark: Apache Spark [24] is a general purpose distributed computing engine that uses a multi-stage, in-memory computational model to achieve fast and scalable performance. Spark core [25] is the central component of the Spark project and it is responsible for creating parallel tasks and scheduling them. Spark SQL [8] is a component on top of Spark core that integrates relational processing with Spark's functional programming API. Users can interact with Spark SQL via different available APIs, including a SQL API and Spark shell, and they can read data from existing Hive tables. Spark SQL uses *Catalyst*, an open-source extensible query optimizer, to analyze query plans and do runtime code generation which eventually gets executed by Spark core.

B. Columnar Storage Formats

The advantages of using columnar data formats are well known for analytical workloads in relational databases [26] [27]. They improve the storage efficiency by effective data compression and achieve significant performance gains by moving only relevant portions of data into memory during query processing. Columnar storage formats have been available for storing data in HDFS [28] for quite some time. Currently, the *Optimized Row Columnar* (ORC) format [19] and the *Parquet* format [29] are the two most popular ones for HDFS. As ORC is mainly optimized for Hive, and Parquet has become the suggested file format for Spark, we chose these two formats to run the TPC-H workload to examine the impact of such optimized storage formats on performance.

Both ORC and Parquet utilize type-specific writers that use different built-in compression techniques to create much smaller files from the original data. Beside compression, these formats achieve I/O efficiency via reorganizing data and splitting it horizontally into fixed size chunks (*stripes* in ORC and *row groups* in Parquet). They also include lightweight indexes that contain statistics such as the minimum and maximum values for each column in each of these chunks. By combining these indices with a filter push-down optimization, the file readers in these columnar formats can sometimes skip entire groups of rows that are not relevant to a query based on the query's predicate.

C. TPC-H Workload

TPC-H [11] is a decision support benchmark that runs against a database with a normalized schema consisting of eight tables. The query suite in TPC-H consists of 22 read-only queries where each tries to answer a specific "business question" in the context of the benchmark. The official specification for TPC-H includes a translation of the functional definitions of the queries in standard SQL. We used this exact set of SQL statements for System-X. For Hive and Spark SQL, we used the revised versions of TPC-H queries in HiveQL that were created by the authors of [15]. This set uses some of the later features in HiveQL such as its support for nested sub-queries. For AsterixDB, we created a translation of the query suite into SQL++ that can be found at [30].

The TPC-H benchmark specification also describes the set of primary and foreign keys (PK/FK) in the database and allows for the creation of auxiliary index structures on these key columns. We added these indices to the schema definitions for AsterixDB and System-X as they each have built-in support for secondary indices.

D. Nested Schema

TPC-H as defined uses a normalized 1NF schema that is native to relational systems. In a 1NF schema, we have exactly one data value within a single row/column position. In contrast, a nested schema allows multiple values per position. When properly used. a nested schema can help an analytical workload achieve significant potential performance gains at scale, as logically related values can be grouped and physically stored together in the same dataset. This can eliminate some of the distributed and potentially expensive PK/FK joins that are common in analytical workloads. With a nested schema, such a join turns into scanning a single dataset in parallel as the child records (FK side of the join) are now stored with their unique parent (PK side). Aggregations, which are also common in analytical workloads, can also benefit from a nested schema performance-wise as the children are already grouped with the parent. On the other hand, a nested schema can negatively impact the scan-dominant parts of an analytical workload since embedding children records into their parents increases the I/O size for full scans.

In this paper, we decided to create a modified version of the schema and queries in TPCH in order to evaluate the actual impact of a nested schema on its performance. For this purpose, we picked the LINEITEM table (which has the largest size among all tables for a given TPC-H scale factor) and nested its rows into the ORDERS table. This means for a given ORDERS record, we now store a list of all LINEITEM records that belong to that specific order. This is natural, as the lineitems are simply "part of" an order in an E-R sense. We called this new dataset NESTEDORDERS. We also modified all of the queries (19 out of 22 queries) that use either of the two tables or both and rewrote them to use the NESTEDORDERS dataset.

We used AsterixDB, the representative of NoSQL systems in our experiments, which has built-in support to store and query nested schemas, to evaluate the nested extension to the benchmark and compare its performance to the original normalized design.

IV. Experiments

In this section, we discuss the details of the experiments. We describe the hardware and the settings used in our tests followed by presenting and analyzing the performance results.

A. Experimental Setup

For our experiments, we used a 10-node IBM x3650 cluster with a Gigabit Ethernet switch. Each node had one Intel Xeon processor E5520 2.26GHz (4 cores), 12 GB of RAM, and four 300GB, 10K RPM hard disks. The machines were running 64-bit CentOS 6.6 as their operating system. Three disks on each machine were used as local persistent storage. The fourth disk was dedicated to storing the transaction and system logs.

We ran the TPC-H workload itself using a client running on a separate machine (with 16GB of RAM and 4 CPU cores) that was connected to the tests' cluster via the same switch. We measured the average end-to-end response time of each individual query from the client's perspective and used it as the performance metric. In each test, we ran the full set of 22 TPC-H queries sequentially, one after the other, three times for each combination of system/settings. The first iteration was used to warm-up the caches and we report the average of the last two runs as the response time for each query.

We used Apache Hadoop version 2.6.0 with a heap size of 8GB and a replication factor of 1. Each node in the cluster was running the DataNode daemon for HDFS and Yarn's NodeManager daemon. The NameNode and ResourceManager daemons were running on a separate machine with a similar configuration as the cluster nodes. For Apache Hive and Tez, we used the stable versions 1.2.0 and 0.7.0 respectively. We enabled optimizations in Hive such as predicate push-down and map-side joins.

We used Apache Spark version 1.5.0 with the Kryo serializer. Each worker was running four executors (one per core) with 2GB of memory. The driver program to execute our Spark benchmarking application ran on the client machine with 6GB of memory and it used Hive's metastore to access the schema and data for the tables created on files in HDFS.

For AsterixDB, we used version 0.8.9 with one NC running on each node with 3 partitions. We set a maximum of 8GB of memory and 2GB of buffer cache size and increased the system's join and sort memory budgets to 128MB per NC.

For System-X, we used a commercial version that was provided to us in 2013 by its vendor. Each node in the cluster was serving three database partitions. The automatic memory manager of System-X was responsible for tuning its resource allocations. A JDBC driver, provided by the vendor, was used by the client to submit the queries to the system.

a) Data and storage: We conducted our experiments with a focus on the scale-up characteristics of the systems. For this purpose, we considered three cluster scales with 9, 18, and 27 partitions running on 3, 6, and 9 machines respectively. We placed 50GB of TPC-H data on each machine, which is almost five times their available memory. To do so, we used the DBGen tool, included in TPC-H benchmark, to generate a total of 150GB, 300GB, and 450GB of data for these different system scales.

For Hive, we used two types of tables: external tables created on raw text data files and ORC format tables. In Spark SQL we considered both external and Parquet tables. For the ORC and Parquet cases, the CUSTOMER and SUPPLIER tables were created as partitioned tables using the NATION_KEY column as the partitioning attribute (there are total of 25 possible values for NATION_KEY). For AsterixDB we loaded data into internal datasets and included the attributes from the TPC-H schema in our DDL statements. We also created secondary indices on the attributes according to the rules in TPC-H specification. In System-X, we created hash-partitioned tables using the primary key(s) in each table. We also created the secondary indices and included the definition of referential constraints (PK/FK relationships) in the DDL statements. After loading data for each scale factor, we ran the statistics gathering scripts of System-X to provide its cost-based optimizer with the information required for generating effective query plans.

B. Results

In this part of the paper, we review the experimental results that we obtained from running all systems on the three different cluster scales.

1) Data size: Table I shows the total size of each table in the TPC-H schema for the scale of 450GB after loading based on the target systems and formats. For AsterixDB and System-X, the number includes the total size of all secondary indices on a specific table. For the smaller scales of 150GB and 300GB, the numbers are not shown as they scaled down proportionally. As one can see, there are significant storage size differences between different formats and systems. ORC and Parquet are the most efficient formats in terms of storage size due to their use of columnar storage and built-in compression. These two formats use different compression techniques and physically organize the data in different manners, which is why they differ from one another in terms of their total size. AsterixDB and System-X use their own managed storage layers. Each of these systems has its own storage format and loads the data into a variant of B-Tree and organizes the records in pages with enough information stored per page for future access. This comes with additional overheads in terms of storage space as compared to the raw data format.

2) Query Times: Tables II, III, and IV show the full set of results that we obtained on the three scales. There were cases where we could not obtain a stable response time to report: For Spark SQL, Q11 took a very long time (on the order of hours) within an aggregation step at all scales, and Q21 faced a failure in memory allocation for the scales of 300GB and 450GB. In Hive, Q4 did not complete with Tez/ORC, as the framework killed a Yarn container after running for a long time. In the tables, the cells corresponding to these cases are marked by "-". We also have three cells per table for *AsterixDB Nested* where the response time is shown in parenthesis (Q2, Q11 and Q16). These are the queries in TPC-H which do not use the

	Nation	Region	Part	Supplier	Partsupp	Customer	Orders	Lineitem	Nested *Orders*
Text	2 KB	389 Bytes	11.07	0.64	55.4	11.12	80.97	363.67	-
ORC	1.75 KB	1 KB	1.77	0.23	13.86	3.68	19	81	-
Parquet	3.2 KB	1.1 KB	5.7	0.67	50.4	10.8	60	154	-
AsterixDB	9.6 MB	1.92 MB	16.39	0.92	70.2	14.71	116.1	621	680
System-X	54 MB	54 MB	13	0.79	57	12	86	412	-

TABLE I
TPC-H TABLES SIZE (IN GB) - SF 450

LINEITEM or ORDERS tables, so their response times are the same for both the nested and the normalized cases.

In Hive, we obtained results for four different combinations by using both the MR and Tez execution engines against the ORC and text formats. We observed that the Tez/ORC combination outperforms other three combinations for almost all queries. For this reason and to save space, here we only include the Tez/ORC and MR/Text results for Hive to show its best performance and its gain compared to the alternative runtime and storage format.

Tables II to IV show that that no system or storage format was able to provide the best performance for all of the queries, and different systems showed different scale-up behaviors.

AsterixDB shows a proper scale-up behavior for both the normalized and nested schema cases, as for most queries the response times remain more or less the same while the data and number of partitions grow. The datasets are stored in AsterixDB's binary data format (ADM) and there is no need for a format transformation as records are being read. Moreover, I/O requests are done in a single-threaded fashion per I/O device (partition) by AsterixDB, which means that readers won't be blocked by other concurrent reading threads on the same partition, and I/O is not really a bottleneck for the majority of such requests. Comparing AsterixDB's performance for pairs of queries between the normalized and nested cases, one can see that while there are some queries that considerably benefit from nested records (such as Q18), there are other queries that did not achieve a measurable performance gain (such as Q19) or even got worse (such as Q22) with nesting. This is expected, as those queries that involve costly joins between the ORDERS and LINEITEM datasets are expected to benefit the most from the nested schema as a result of join elimination. On the other hand, scan-dominant queries on the ORDERS dataset experience longer scan times as a result of reading more data. We will look more closely at example queries from both groups in the next section.

In Spark SQL, Parquet tables outperform the raw text format for all queries and show a reasonable scale-up behavior and better performance compared to the other systems for several queries. This is mainly due to Parquet's built-in optimizations as a columnar storage format combined with Spark's pipelined in-memory computation model. Unlike the raw text format, with Parquet we observed parallel reads happening among all Spark workers. In addition, Parquet's effective data compression and built-in metadata indices reduce the I/O effort significantly both by reading less data and by skipping unnecessary column chunks according to the query.

Similar to Spark SQL, Hive on Tez/ORC leverages optimizations in Tez (as a parallel distributed execution engine) and ORC (as a compressed columnar storage format) and achieves a satisfactory level of scale-up. Comparing to Hive's MR execution engine, the queries benefited from Tez, especially those involving multi-way joins. The small job-starting overhead and skipping of intermediate materialization steps in Tez as compared to MR reduce the overall join costs. The combination of Tez/ORC also showed better behavior in terms of utilizing the Map-side join optimization in Hive. There were several queries that used Map-side joins only with this combination (e.g., Q2 and Q5).

System-X shows a nearly flat (ideal) scale-up behavior for most of the TPC-H queries. There are two queries for which System-X had a strange behavior: Q10 shows a non-linear performance change between different scales, and Q22's response time grows as the total data size grows. We will look at these two cases in more detail in the next Section.

V. SELECTED QUERIES

In this section, due to limited space, we examine a selected handful of queries in greater detail. The main reason to choose these particular queries is that either the systems performed significantly differently for them or a specific system showed a different scale-up behavior as compared to the other queries.

A. *Query 1*

The first query from the TPC-H workload provides a summary pricing report for all lineitems shipped as of a given date [11]. It accesses only one, but the largest, table in the database, namely LINEITEM. From a runtime perspective, this is a query that can measure how fast a system performs filtered scans and aggregations on a large table. The size and format of storage along with the way that selection predicates are evaluated on records are the key factors in determining the systems' response times for this query.

As Table IV shows for the largest scale (SF=450GB), Spark SQL with Parquet was able to achieve the best time among all systems. The execution of this query in Spark consists of two stages: during the first one, the LINEITEM table is scanned with the selection predicate and a local aggregation is performed. After a data exchange, the global aggregation is

	AsterixDB *Normalized*	AsterixDB *Nested*	Spark SQL *Text*	Spark SQL *Parquet*	Hive *MR & ORC*	Hive *Tez & ORC*	System-X
Q1	603	654	827	154	315	273	270
Q2	100	(100)	162	73	332	141	56
Q3	474	302	983	263	695	489	417
Q4	407	288	842	114	754	-	409
Q5	445	258	1212	500	1155	891	421
Q6	317	338	695	82	84	90	259
Q7	932	492	1256	385	2799	1044	590
Q8	497	312	1196	358	1023	1061	375
Q9	1921	1542	1929	1057	3607	3523	587
Q10	381	298	901	168	532	359	1604
Q11	79	(79)	-	-	534	228	41
Q12	432	370	877	123	315	265	320
Q13	202	352	244	133	426	280	99
Q14	276	372	687	86	166	142	442
Q15	276	364	1366	165	360	186	319
Q16	158	(158)	158	44	278	147	33
Q17	1354	1318	1711	445	1582	1716	269
Q18	706	314	1612	218	2018	1702	375
Q19	395	425	827	158	1302	1394	265
Q20	453	496	823	137	474	319	2316
Q21	2705	2160	2575	725	2093	1649	8776
Q22	282	1737	181	42	337	203	336

TABLE II
TPC-H QUERIES RESPONSE TIME (IN SEC) - SF 150

	AsterixDB *Normalized*	AsterixDB *Nested*	Spark SQL *Text*	Spark SQL *Parquet*	Hive *MR & ORC*	Hive *Tez & ORC*	System-X
Q1	602	660	1863	162	320	277	277
Q2	151	(151)	246	76	343	149	58
Q3	505	315	2381	491	705	518	429
Q4	410	291	2042	118	756	-	415
Q5	590	304	2859	930	1290	982	433
Q6	317	340	1742	86	91	93	262
Q7	982	519	2804	635	3041	1127	698
Q8	536	319	2396	388	1112	1089	452
Q9	2012	1671	3354	1152	3925	3629	595
Q10	418	333	2231	298	534	378	1618
Q11	80	(80)	-	-	781	246	42
Q12	435	383	2166	220	325	273	340
Q13	211	374	422	141	453	292	116
Q14	281	375	1697	92	177	144	446
Q15	284	370	3397	189	360	189	328
Q16	173	(173)	225	69	334	150	44
Q17	1391	1393	3767	475	1642	1757	271
Q18	768	325	3534	244	2060	1751	386
Q19	442	494	1896	179	1324	1489	269
Q20	454	498	1960	140	531	331	2914
Q21	2734	2223	-	-	2274	1845	11102
Q22	328	1915	349	45	357	210	868

TABLE III
TPC-H QUERIES RESPONSE TIME (IN SEC) - SF 300

	AsterixDB *Normalized*	AsterixDB *Nested*	Spark SQL *Text*	Spark SQL *Parquet*	Hive *MR & ORC*	Hive *Tez & ORC*	System-X
Q1	605	658	2870	168	324	283	286
Q2	189	(189)	326	80	351	152	60
Q3	513	320	3825	709	714	537	436
Q4	412	294	3236	121	759	-	424
Q5	719	348	4512	1359	1385	1082	442
Q6	320	341	2717	90	96	98	265
Q7	1091	592	4272	879	3261	1218	796
Q8	540	328	3599	417	1190	1104	524
Q9	2069	1777	4785	1260	4423	3745	601
Q10	428	352	3558	417	535	389	455
Q11	81	(81)	-	-	1025	266	42
Q12	439	400	3451	313	339	285	355
Q13	228	382	573	149	474	300	125
Q14	291	379	2702	97	186	146	454
Q15	296	380	5419	210	360	193	334
Q16	192	(192)	290	90	385	154	51
Q17	1414	1408	5827	504	1695	1784	272
Q18	805	327	5417	273	2105	1792	399
Q19	509	533	2962	201	1341	1573	273
Q20	454	503	3097	141	575	346	3535
Q21	2765	2285	-	-	2417	2016	12986
Q22	347	1958	513	47	268	213	1393

TABLE IV
TPC-H QUERIES RESPONSE TIME (IN SEC) - SF 450

computed in the second stage. Effective I/O parallelization in conjunction with Parquet readers' ability to skip non-relevant blocks of data according to the query's range filter are the main reasons in achieving that level of performance. Similar to Spark SQL's stages, Hive uses two MapReduce jobs to process this query. Using Tez and the ORC file format, the same argument as the one for Spark SQL applies to Hive.

In both AsterixDB and System-X, we have a secondary index on the column used in the query's predicate, i.e., L_SHIPDATE. The selectivity of this predicate is relatively large. For the scale of 450GB, it selects more than 887 million tuples out of almost 2.7 billion LINEITEM tuples. As a result, exploiting this auxiliary index, instead of a full sequential scan, will not help performance. System-X's cost-based optimizer decides to fully scan all LINEITEM records for this reason. In AsterixDB, the "skip-index" hint is available for this purpose and it was used for the response times we report. Moreover, the combination of values in the grouping columns (L_RETURNFLAG and L_LINESTATUS) has only 4 possible values. In AsterixDB we used hash-based aggregation (enabled also with a hint) for the query and it resulted in better performance as compared to sort-based aggregation. (It reduced the response time for SF=450GB by about 30%). However, System-X's optimizer decided to use sorting for grouping and aggregation here. The first phase of the query (fully scanning LINEITEM) is comparable for both systems (for SF=450GB, it took 270 seconds for AsterixDB and 250 seconds for System-X). However, System-X shows better performance during the second phase (filtering, grouping and aggregation) and ends up performing better than AsterixDB in terms of the overall performance on this query.

This query took longer with the nested schema compared to the normalized one for AsterixDB, as the sequential scan needs to read a larger dataset (i.e. NESTEDORDERS). While the total number of records is the same in both cases, the actual amount of data being read is more with nesting since the data related to the ORDERS portion of each record will be fetched as well. The ratio of total response time between the nested and normalized cases is proportional to the ratio of the sizes of the NESTEDORDERS and LINEITEM datasets.

B. Query 10

This query finds the top 20 customers in terms of their effect on lost revenue for a given quarter [11]. The query involves a 4-way join between the CUSTOMER, ORDERS, LINEITEM, and NATION tables, which have a wide range of sizes, with LINEITEM and ORDERS being the two largest tables while NATION has only 25 rows. For such a case, an optimizer has different join ordering and join strategy choices to choose from. The systems we picked generate different plans for the query: Spark SQL first joins CUSTOMER with filtered ORDERS tuples and then joins these results with filtered LINEITEM tuples and finally joins the results with the NATION table. AsterixDB, with the normalized schema, joins CUSTOMER and filtered ORDERS tuples first (while it uses the secondary index on O_ORDERDATE for applying the predicate on ORDERS). This join is followed by joining its results with the NATION table. Finally the results are joined with the filtered LINEITEM tuples. In terms of the join strategy, AsterixDB currently uses hybrid hash joins for its equi-joins. Spark SQL, on the other hand, decides to use sort merge joins for the first two joins, and for the join involving the NATION table it broadcasts this table and performs a hybrid hash join in each partition. In AsterixDB, switching to the nested schema eliminates the most expensive join (between LINEITEM and ORDERS) and has a significant impact on the response time. As Table IV shows, AsterixDB with nesting achieves the best performance among all systems for the largest scale by performing a scan on the NESTEDORDERS dataset and eliminating the join. Hive follows a similar strategy as Spark SQL and uses 4 map and 4 reduce steps. It first joins CUSTOMER and ORDERS using sort merge join and then replicates NATION and performs a broadcast hash join with LINEITEM.

One reason we found this query interesting is that System-X scaled up non-linearly for SF=450GB as it changed the query plan for this scale. For all 3 scales, System-X first joins filtered ORDERS and LINEITEM tuples. It exploits the secondary indices on O_ORDERDATE and L_ORDERKEY for filtering and joining. For the two smaller scales (150GB and 300GB), System-X proceeds by scanning the CUSTOMER table and joining it with NATION, and as the output is already sorted on customer keys, it chooses a merge join to do the join between the results of the latter join (CUSTOMER and the NATION) and the results of the first join (ORDERS and LINEITEM). In 450GB case, however, System-X decides to broadcast the NATION table across all partitions and do a hash join at the end. This change in the strategy helps System-X achieve much better performance for this query for SF=450GB.

C. Query 18

This query ranks customers based on their having placed a "large quantity" order, which is an order whose total quantity is above a certain level (for this query, SUM(L_QUANTITY)>300) [11]. For the normalized schema, where LINEITEM and ORDERS records are stored in separate tables, all four systems decide to use similar strategies with some small differences. They scan the three tables in the query (namely CUSTOMERS, ORDERS and LINEITEM) and use three joins. They first join CUSTOMERS and ORDERS, then join between these results and grouped/aggregated and filtered LINEITEM records (to apply the "large quantity" predicate); the last join is between the results of the second one and LINEITEM. Projected records from the results of the third join feed an aggregation to produce the final results.

Spark SQL and Hive hash exchange all three tables on the join keys and do sorting to use merge joins. System-X uses the secondary indices on the primary/foreign keys to run index nested loop joins. AsterixDB hash exchanges LINEITEM and ORDERS records and runs hybrid hash joins.

One reason we consider this query to be interesting is the considerable performance gain achieved using the nested schema. With nesting, all LINEITEM records that belong to a certain order are already grouped and stored within their parent ORDERS record. Therefore, the joins between the LINEITEM and ORDERS tables in the normalized case to apply the "large quantity" predicate can be replaced by a predicate evaluation step during scanning of the NESTEDORDERS dataset. AsterixDB does that using a sub-plan (involving unnesting and aggregation on the list of lineitems per record) when scanning NESTEDORDERS. The projected results of this filtered scan are then hash exchanged and joined with CUSTOMERS to produce the final results. Tables II to IV show that such a change in plan to eliminate two costly joins helps AsterixDB to achieve better performance (more than two times faster) compared to the normalized schema.

D. Query 19

This query finds the gross revenue for all orders shipped by air and delivered in person for three types of parts [11]. The query joins the PART and LINEITEM tables on their PK/FK columns while it has complex disjunctive and conjunctive predicates on both tables. A careful look at these filters shows that the predicates on the L_SHIPMODE and L_SHIPINSTRUCT columns from LINEITEM are common among all three disjuncts. These predicates in fact are strongly selective, as approximately only 3.5% of tuples pass them (in SF=450GB, less than 97 million tuples pass them out of more than 2.7 billion total LINITEM tuples). Therefore, pushing these predicates down and applying them prior to the join can help a system's performance significantly.

Spark SQL indeed extracts the common predicate and applies it prior to the join step. It also extracts the common lower bound of the range filter on the P_SIZE column from the PART table and pushes it down, but since all the PART tuples satisfy this lower bound that is of no help in this case. The remaining case-specific predicates on the P_BRAND column from the PART table are applied to the results of the join, which are then fed to the final aggregation step. The plan that System-X generates for this query is similar to Spark's plan; the main difference is in the predicates it picks to push down. System-X decides to apply all of the predicates (both the common and case-specific ones) for both tables while scanning them. The case-specific predicates are grouped and are first applied disjunctively and then applied again in a case-specific manner on the join results. In this way, System-X tries to reduce the input sizes for the join. For example, in SF=450GB, about 58 million (out of more than 2.7 billion) LINEITEM tuples and about 216,000 (out of 90 million) PART tuples pass these filters. Unlike Spark SQL and System-X, Hive does not extract the common predicate on LINEITEM from the disjuncts to push it down. Hive first uses two maps to fully scan both tables and sorts them on the PARTKEY column. Then two reducers are used to perform a merge join, grouping and aggregation. The negative impact of joining a large number of unneeded tuples is the main reason that Hive's performance for all scales are significantly worse than others.

In AsterixDB, the optimizer did not factor out the common predicates to push them down when the query was based on a straightforward translation from SQL. It decided to first fully join LINEITEM and PART, and since the equi-join attribute from the PART side is PART's primary key, LINEITEM is hash-exchanged. This decision results in a significant overhead in data movement and impacts AsterixDB's performance negatively. A more careful translation of the query removes this overhead by helping the system to push down the common predicates and improves its performance by a factor of 3. The reported numbers are based on this revised version. As the ORDERS dataset is not used in this query, switching to a nested schema does not help. In fact, the query's response time even gets slightly worse since scanning the NESTEDORDERS dataset takes longer than scanning LINEITEM as a result of fetching the unneeded ORDERS portion of each record.

E. Query 22

This query counts the number of customers within a specific range of country codes that have not placed orders but have a greater than average positive account balance [11]. This query is interesting to us for two reasons: First, System-X shows its worst scale-up behavior among all queries for this one. Second, switching to the nested schema degrades AsterixDB's performance significantly.

Checking the query plan that each system uses, we observed that calculating the average account balance (i.e. AVG(C_ACCTBAL)) on the CUSTOMER table is the same among all systems. The main performance difference between the systems is based on how the "greater-than" predicate (C_ACCTBAL $>$ AVG(C_ACCTBAL)) and the "anti-join" part (CUSTOMER tuples with no orders) are evaluated. Spark SQL decides to use a sort-merge outer join to find customers with no orders and then applies the greater-than predicate after doing a Cartesian product. Hive uses three map and four reduce steps to process this query. It first calculates the average value (using one map and one reduce) and broadcasts it. Another map step finds the qualified CUSTOMER tuples, and then by using a map and reduce step a group-by on ORDERS based on O_CUSTKEY is calculated. Then Hive performs an outer merge join on the qualified CUSTOMERS and the grouped ORDERS. The greater-than predicate is also applied in this step. The final aggregation happens on the output of this join and generates the results. System-X, however, decides to use the index on O_CUSTKEY on the ORDERS table to evaluate a sub-query for finding customers with no orders; the greater-than predicate evaluation happens in a subsequent nested loop join. In the normalized case, AsterixDB first applies the greater-than predicate and finds the customers with above average balances by performing a nested loop index join and then it evaluates the anti-join sub-query as the last step.

By breaking the query down into pieces and running each one, we realized that the anti-join step to find customers with no orders takes the most time. While both Spark SQL and AsterixDB use outer joins for this part, System-X chooses to use the secondary index on the foreign key column. In each scale, almost 28% of the customer records pass the predicate on C_PHONE and one third of these customers have no orders. Therefore, these predicates have a relatively large selectivity for secondary index lookups. As the data grows, the number of qualifying customers according to the predicates also grows proportionally. Because of the access method that System-X uses to find them (index lookups rather than an outer join), as the cardinality scales up more and more random I/Os are needed, proportional to the data size. This negatively impacts System'x overall response time for all the scales.

Considering the nested schema, since the LINEITEM data is not used in the query, AsterixDB uses the same query plan as the normalized case. However with nesting, fetching the ORDERS records needs to happen by scanning the NESTEDORDERS dataset which is much larger than the ORDERS dataset because of the embedded list of LINEITEM records in each order. Table I shows that for SF=450GB, NESTEDORDERS is 5.8 times larger than ORDERS in size. Comparing AsterixDB's performance for the normalized and nested cases in Table IV shows that a similar ratio exists between the corresponding response times for this query.

VI. Discussion

In this section, we summarize the main lessons that we have learned from our TPC-H based performance study. These lessons are not entirely new, but they provide valuable insight into the current state of reality in Big Data analytics platforms.

L1. Importance of cost-based optimization: As Big Data analytics queries tend to be complex, the quality of the physical plans for processing them plays a significant role in the overall performance. Section 5 showed that the systems considered here come up with different query plans for most of our selected queries (for example Q10 and Q22). In practice, finding the optimal plan is a complicated task, as a complex query may have many different possible execution paths. Having a comprehensive set of rewrite rules, an accurate cost model, statistics about the data, and reasonable estimates of the sizes of intermediate results are crucial factors in generating an efficient plan. From this perspective, relational databases still have significant advantages over modern Big Data systems. However, query optimization in new Big Data platforms is getting richer fast, as they are learning from DBMSs and also focusing on tackling various performance issues that have arisen with Big Data.

L2. Importance of storage format and I/O parallelism: Our experiments clearly showed the beneficial impact of optimized storage formats on Big Data analytics platforms' performance. Both Spark SQL and Hive showed significant performance improvements when switching from the raw text to columnar formats. A part of these performance gains comes from the reduced storage sizes with Parquet and ORC because of data compression. Another key reason is the combination of format-specific readers with the inherent optimizations of these newer storage formats (such as using a columnar model with built-in auxiliary information). Such a combination can reduce the overhead of I/O by avoiding reading unneeded data for a query. The developers of Spark and Hive have put considerable efforts into exploiting the features in these file formats, so by pairing them with other optimizations (such as predicate push-down), I/O becomes less of a bottleneck in the overall performance.

L3. Importance of dataflow processing: Our experiments re-confirmed the performance advantages of using a pipelined query execution model for complex queries whose processing consists of several stages. Hive's results on Tez experienced performance gains for all queries due to the overall impact of reduced scheduling overhead and data materialization. Both Spark and Hyracks [23] (used by AsterixDB) are execution engines built from scratch to run dataflow jobs efficiently. While MapReduce is fine for one-pass computations, especially large computations whose running times require failure-tolerance, it suffers from performance-related issues for more complex jobs due to its limited (stylized) two-step programming model and its spilling of intermediate results to disk.

L4. Denormalized schema vs. normalized schema: Unstructured and semi-structured data are now popular in the Big Data world. Big Data platforms are expected to ingest, store and process large volumes of heterogeneous records in a near real time fashion. These records originate from various sources, including non-relational systems or applications which widely use non-SQL programming languages. Therefore, data items tend to have various complex structures (with fields of different types, including "collections" or multiple levels of nesting). A relational system needs to break a complex record and store it across normalized tables. While normalization comes with advantages, such as having compact records per table with no duplicates or redundancy, it can suffer from performance degradation due to expensive joins when retrieving tuples from different tables to reconstruct the original records. On the other hand, a NoSQL system that allows a nested schema can group the fields in a record tightly together and store them in their original form. In addition, relaxed schema assumptions in the "document" model enable such a system to store related, but heterogeneous, records together in a single dataset. As a result, analytical workloads can benefit from keeping the logical representation and physical storage of data closer to each other, as some expensive distributed joins now turn into scans and aggregations run on pre-grouped values. This means less I/O and less network overhead for some parts of an analytical workload. Scan-dominant parts of the workload, however, can take longer, as they may need to fetch and process more data. Combining nested schema support with columnar formats can potentially improve the latter issue. In our experiments, comparisons of AsterixDB's results for the normalized and nested schema showed examples of both cases (for example, Q18 benefited from nesting, while Q22 got worse).

VII. Conclusion

In this paper, we have reported on a performance evaluation of Big Data systems for OLAP-style workloads. We used the TPC-H benchmark to evaluate four Big Data platforms: Hive, Spark SQL, AsterixDB, and System-X (a parallel commercial RDBMS). We explored several different storage formats for Spark SQL and Hive. We also considered a variant of the benchmark's schema with nesting for AsterixDB. Our results showed that no system, storage format, or schema variant gave the best performance for all of the queries. Moreover, different systems showed different scale-up behaviors. We also showed how the optimized columnar storage formats (ORC and Parquet) or a nested schema can improve performance in many cases. To better understand the performance differences between the systems, we analyzed a selected subset of the queries in more detail to study the impact of some of the systems' different optimizations on performance.

Acknowledgement. We would like to thank certain individuals affiliated with System-X's vendor for their valuable help on this work. We thank Avrilia Floratou for sharing the revised Hive queries from [15]. We also thank Yingyi Bu for his help on revising and improving some of the queries for AsterixDB.

References

[1] "Forbes Report," http://onforb.es/1nwCOsb/.
[2] A. Edwards and G. Davies, "Understanding Analytic Workloads - Meeting the complex processing demands of advanced analytics," 2011.
[3] "Apache Hive," http://hive.apache.org/.
[4] "Presto," http://prestodb.io.
[5] "Impala," http://impala.io/.
[6] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *PVLDB*, 2010.
[7] "Apache Drill," https://drill.apache.org.
[8] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational data processing in Spark," in *SIGMOD*, 2015.
[9] "Big SQL," https://www.ibm.com/us-en/marketplace/big-sql.
[10] M. J. Carey et al., "AsterixDB: A Scalable, Open Source BDMS," *PVLDB*, 2014.
[11] "TPC-H," http://www.tpc.org/tpch.
[12] "TPC-DS," http://www.tpc.org/tpcds/.
[13] 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.
[14] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang, "Can the Elephants Handle the NoSQL Onslaught?" *PVLDB*, 2012.
[15] A. Floratou, U. F. Minhas, and F. Özcan, "SQL-on-Hadoop: Full circle back to shared-nothing database architectures," *PVLDB*, 2014.
[16] 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.
[17] https://www.slideshare.net/cloudera/data-modeling-for-data-science-simplify-your-workload-with-complex-types-in-impala.
[18] "Apache Hadoop," http://hadoop.apache.org/.
[19] "Apache ORC," https://orc.apache.org/.
[20] "Apache Tez," https://tez.apache.org.
[21] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications," in *SIGMOD*, 2015.
[22] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li, "Storage Management in AsterixDB," *PVLDB*, 2014.
[23] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing," in *ICDE*, 2011.
[24] "Apache Spark," http://spark.apache.org.
[25] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *USENIX conference on Hot topics in cloud computing*, 2010.
[26] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems," *PVLDB*, 2009.
[27] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil *et al.*, "C-store: a column-oriented DBMS," in *VLDB*, 2005.
[28] Apache HDFS, http://hadoop.apache.org/hdfs/.
[29] "Apache Parquet," https://parquet.apache.org/.
[30] https://github.com/pouriapirz/tpch-on-asterixdb.