BigFUN: A Performance Study of Big Data Management System Functionality

Pouria Pirzadeh
University of California Irvine
Irvine, USA
Email: pouria@uci.edu

Michael J. Carey
University of California, Irvine
Irvine, USA
Email: mjcarey@ics.uci.edu

Till Westmann
Couchbase [§]
Mountain View, USA
Email: till@couchbase.com

***Abstract*—In this paper, we report on an evaluation of four representative Big Data management systems (BDMSs): MongoDB, Hive, AsterixDB, and a commercial parallel shared-nothing relational database system. In terms of features, all offer to store and manage large volumes of data, and all provide some degree of query processing capabilities on top of such data. Our evaluation is based on a micro-benchmark that utilizes a synthetic application that has a social network flavor. We analyze the performance results and discuss the lessons learned from this effort. We hope that this study will inspire future domain-centric evaluations of BDMSs with a focus on their features.**

I. Introduction

The IT world is excited about the Big Data ”buzz”. Immense volumes of data are generated continuously in different domains and there is clear merit in analyzing and processing this data. New emerging platforms for this purpose can be largely categorized into two groups: interactive request-serving systems (NoSQL), mainly serving OLTP types of workloads with simple operations, and Big Data analytics systems, which process scan-oriented OLAP types of workloads. The variety of Big Data systems makes it difficult for end users to pick the most appropriate system for a specific use case. In this situation, benchmarking Big Data systems can help provide more insight by offering a better understanding of the systems and also obtaining a set of guidelines to make the correct decision in picking a system for a specific application. The performance evaluation of Big Data systems is thus a challenging, but essential, need in today’s Big Data world. While well-established, comprehensive benchmarks exist to evaluate and compare traditional database systems, benchmarking efforts in the Big Data community have not yet achieved maturity. Some of the performance reports and white papers only show ’hand-picked’ results for scenarios where a system is behaving just as desired. In addition, there are still ongoing debates about what the correct set of performance metrics is for a Big Data system. These issues make the evaluation and comparison of these systems complicated.

There are a few major Big Data benchmarking exercises that have gotten serious attention from the community. These efforts have mostly kept their focus on a well-defined, but narrow, domain of systems or use cases. Examples are YCSB [1] for OLTP-like and the reported study in [2] for OLAP-like use-cases. Additionally, the types of operations that these efforts have included do not cover the full set of functionality that a complete Big Data system should offer.

In this paper, our aim is to contribute to the Big Data performance study area from a slightly different angle. We study and compare Big Data systems based on their available features. We use a micro-benchmark, BigFUN (for Big Data FUNctionality), that utilizes a simple synthetic application with a social network flavor to study and evaluate systems with respect to the set of features they support along with the level of performance that they offer for those features. The BigFUN micro-benchmark focuses on the data types, data models, and operations that we believe a complete, mature BDMS should support. We report and discuss initial results measured on four Big Data systems: MongoDB [3], Apache Hive [4], a commercial parallel database system, and AsterixDB [5]. Our goal here is not to determine whether one Big Data management system (BDMS) is superior to the others. Rather, we are interested in exploring the trade-offs between the performance of a system for different operations versus the richness of the set of features it provides.

We should emphasize that this paper is a first step towards the goal of studying Big Data systems feature-wise. While we did our best in designing BigFUN and using it to evaluate a set of representative platforms, we do not claim that this work is comprehensive. We hope our work can show the merit of the direction it has taken in Big Data benchmarking, and we expect future work to expand on this effort.

II. Related Work

The history of benchmarking data management technologies goes back to the 1980s, when the first generation of relational DBMSs appeared. The Wisconsin benchmark [6] and the Debit-Credit benchmark [7] are among the first works in this area. Because of their major influences on data management systems, the Transaction Processing Performance Council (TPC) came into existence and developed a series of benchmarks such as TPC-C and TPC-H. A number of benchmarks emerged from the research community. Examples include: OO1 [8], OO7 [9], BUCKY [10], and XMark [11].

[§]Work done while at Oracle Labs.

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

With the emergence of Big Data, work on Big Data benchmarks has begun to appear. This work can roughly be divided into two main groups. The first addresses the problem of evaluating Big Data serving technologies from a broad perspective. BigBench [12] describes an end-to-end Big Data benchmark proposal. A recent overview paper [13] discusses potential pitfalls and unmet needs in Big Data benchmarking.

The second category of work on Big Data benchmarking mainly proposes a new Big Data benchmark and presents the results obtained by running it. For Big Data analytics, [2] was the first work to compare Hadoop against Vertica and a row-organized parallel RDBMS (the same DBMS examined here) using a workload consisting of different selections, aggregations, and a join. On the NoSQL front, YCSB [1] (from Yahoo!) presented a multi-tier benchmark that uses mixed workloads of short read and write requests against a number of key-value stores and sharded MySQL. Other works, such as [14] and [15], have extended this effort further. Recently [16] looked at both OLAP and OLTP workloads using existing benchmarks. Other examples of works include LinkBench [17], GridMix [18], PigMix [19], and BG [20].

While the Big Data community has identified the merit, major obstacles and challenges in evaluating Big Data systems, there is still a long way (and potential opportunities) to go to create practical benchmarks that can be widely adopted and used by users and the industry.

III. Systems Overview

In this section, we provide a brief overview of the systems that we use in our evaluation here. We picked them because most have been used extensively for different use cases. Moreover, their rich set of features makes them reasonable candidates for our performance study.

System-X: System-X is a commercial, parallel, shared-nothing, relational DBMS. It defines its schemas using the relational data model, partitions data horizontally, and manages storage using native RDBMS storage technology. It supports different types of indices. A client can submit queries through the system's supported APIs, such as standard JDBC drivers. System-X has a mature cost-based query optimizer to convert an input SQL query into an optimized query plan that is then executed in parallel on the cluster. System-X represents the traditional way of managing Big Data.

Apache Hive: Hive [4] is a data warehouse that provides a SQL-like interface (HiveQL) on top of Hadoop. It supports various file formats for its tables that have significant performance differences, e.g., sequence files, RCFile, ORC, and Parquet. A client submits a query through APIs such as Hive CLI or HiveServer2. A query is compiled into an optimized execution plan of map and reduce jobs. These are executed by the Hadoop framework, and the results can be stored in HDFS or delivered back to the user. Hive represents the class of batch-oriented Big Data analytics platforms.

MongoDB: MongoDB [3] is a NoSQL document database that stores its data in schema-less collections using BSON. It supports automated sharding and load balancing to distribute data on cluster. MongoDB also supports various types of indices and operations. In MongoDB, aggregations can be done through an aggregation framework as well as MongoDB's mapReduce command. Clients directly connect to MongoDB processes to submit requests. Queries run against a single collection, and there is no support for joins. Users need to change their data model (using embedded documents) or perform client-side joins for that purpose. MongoDB represents the current class of NoSQL document stores.

AsterixDB: AsterixDB [5] is a new open source Big Data management system for storing and processing semi-structured data. It has its own declarative query language (AQL) and data model (ADM). ADM is a "super-set" of JSON, with additional data types compared to JSON. AsterixDB supports its own native storage (hash-partitioned LSM B+ Trees) along with external storage (currently HDFS). It has support for different types of indices such as B+ Trees, spatial indices, and text indices. A built-in data feed service for continuous data ingestion is another of its features. As the execution engine, AsterixDB uses Hyracks, a data-parallel runtime platform for shared-nothing clusters. Clients use an HTTP-based API to submit queries. A query is compiled and optimized by a rule-based optimizer and executed as a Hyracks job whose results are delivered back to the client. AsterixDB represents a new generation of Big Data management platforms.

IV. Data and Workload Description

A mature Big Data system must manage huge volumes of data coming from multiple sources with different schemas, along with high rates of incoming new data and updates. The design of the BigFUN micro-benchmark aims at reflecting these expectations in its data and workload. We followed the idea of the Wisconsin benchmark [6] and designed a generic synthetic database populated with randomly generated flexible records that can be scaled accurately. We designed the BigFUN schema to cover a wide range of basic and rich data types. The workload consists of various simple and complex operations to study the level of support and performance for a range of different features and functionality in a given system. In this section, we introduce the database in BigFUN along with the operations in its workload.

A. Database

The BigFUN schema includes simple, complex, and nested data types along with unique and non-unique attributes that can serve different indexing and querying purposes. In BigFUN, we are interested in identifying the supported data types in a system, including richer data types such as temporal or spatial, and explore the level of support for secondary indices on them. We also want to check if a system can store heterogeneous records and records with nesting or if it requires all records in a dataset to have the same schema and be normalized.

Data Types: We store information about two imaginary social networks in our database: *Gleambook* and *Chirp*. We use five data types, two of which are nested in others (Figure 1):

1) GleambookUserType: Captures information about the

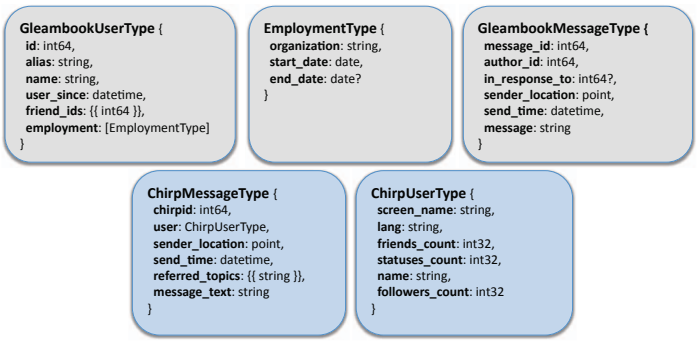

Fig. 1. Nested BigFUN schema

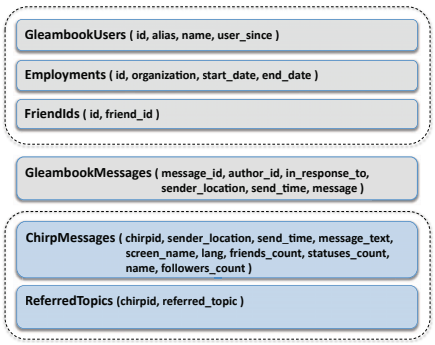

Fig. 2. Normalized BigFUN schema

Gleambook network users. With basic information such as the id and name, each user's record contains a list of *EmploymentType* records showing the employment history and a set of *friend_ids* for the ids of other users connected to this user. *2) GleambookMessageType:* Contains information about a message in the Gleambook network. The message text is stored in the *message* attribute and *author_id* is a (foreign key) attribute that shows the sender of a message using his id. *3) ChirpMessageType:* Captures information about a chirp message in the Chirp network. It includes *user* as a nested attribute of type *ChirpUserType* to store information about the sender. *Referred_topics* is an unordered list of keywords storing the hash tags of a message.

The attributes in Figure 1 are a super-set of the attributes that a given instance of each type may have. Optional attributes are denoted by '?', such as *end_date* in *EmploymentType*. In a system that supports semi-structured data, optional attributes can simply be left out of records that they do not exist in. If a system only supports structured data, where all attributes appear in each record, missing attributes are set as 'NULL'.

Datasets: The number of datasets in an implementation of BigFUN depends on whether records can be stored with nesting or need to be normalized. Using a rich data model, the BigFUN records can be stored in three datasets: *GleambookUsers*, *GleambookMessages* and *ChirpMessages*, with each dataset storing records of one of the top-level data types in Figure 1. But if a system only supports flat records, the schema needs to be normalized. One way of doing that is using six datasets (Figure 2). While the *GleambookMessages* dataset remains intact, the *employment* and *friend_ids* attributes in *GleambookUsers* records and *referred_topics* in *ChirpMessages* records are stored in separate datasets, and the *id* of the *GleambookUsers* and *chirpid* of *ChirpMessages* are used as the parent key in these child datasets.

Secondary Indices: The BigFUN micro-benchmark uses secondary indices in a system when justified and the system under test has support for them. This includes B-Trees and spatial and text indices. More details are provided later in section V.

Data Generation: We created a data generator for BigFUN that uses a scalable approach to generate synthetic data based on a given scale factor. The scale is interpreted as the total size of data (in bytes) across all datasets to characterize one specific load. The generator has a number of adjustable parameters and policies to control various properties of the synthetic data. For temporal and spatial attributes, it selects the values from defined intervals and regions. It generates synthetic, yet "meaningful", messages using a set of message templates and a dictionary of keywords with different expected popularity. It also injects some noise, in the form of misspellings, as potential human errors in editing messages. More details about the data generator can be found in [21].

B. Workload

The BigFUN micro-benchmark's workload consists of the operations shown in Figure 3. The first six groups contain read-only queries, categorized based on the functionality and features that each explores. The last group contains data modification operations. Table I lists these queries and operations.

Our read-only queries try to meet these goals:

Clarity: One should be able to associate each query with a user request that could arise in a real Big Data application. For example, the "unique record retrieval" query (Q1) can be mapped to fetching the profile of a user in a social network.

Simplicity: Each query should be an independent operation that evaluates a well-defined and reasonably small set of features. For example, the "global aggregation" query (Q6) measures the performance of aggregating the results of applying a built-in function on selected records.

Coherence: A relationship exists between groups of queries, such that comparing results among them reveals more insight about the performance of a system. For example, Q6 performs a simple aggregation, while Q7 and Q8 each add an extra step on top of that.

We discuss each group of operations in more detail below:

G1. Single Dataset - Simple: The first group of operations consists of three queries (Q1 to Q3) which retrieve full records in the GleambookUsers dataset using predicates (with varying selectivities) on the primary key (*id*) or a non-unique temporal attribute (*user_since*). If the schema is normalized, a system needs to access more than one dataset to fetch all the attributes of the retrieved records and combine them together.

G2. Single Dataset - Complex: This group of operations considers quantification and aggregation. The quantification queries (Q4, Q5) use *end_date* in the employment attribute

QId	Name	Description
Q1	Unique record retrieval	Retrieve an existing user using his or her user id.
Q2	Record id range scan	Retrieve the users whose user ids fall in a given range.
Q3	Temporal range scan	Retrieve the users who joined the network in a given time interval.
Q4	Existential quantification	Find basic and employment information about users who joined the network in a given time interval and are currently employed.
Q5	Universal quantification	Find basic and employment information about users who joined the network in a given time interval and are currently not employed.
Q6	Global aggregation	Find the average length of chirp messages sent within a given time interval.
Q7	Grouping & aggregation	For chirp messages sent within a given time interval find the average length per sender.
Q8	Top-K	Find the top ten users who sent the longest chirp messages (on average) in a given time interval.
Q9	Spatial selection	Find the sender names and texts for chirp messages sent from a given circular area.
Q10	Text containment search	Find the texts and send-times for the ten most recent chirp messages containing a given word.
Q11	Text similarity search	Find the texts and send-times for the ten most recent chirp messages that contain a word similar (based on edit distance) to a given word.
Q12	Select equi-join	Find the users' names and message texts for all messages sent in a given time interval by users who joined the network in a specified time interval.
Q13	Select left-outer equi-join	For users who joined the network in a given time interval, find their name and the set of messages that they sent in a specified time interval.
Q14	Select join with grouping & aggregation	For users who joined the network in a given time interval, find their ids and the total number of messages each sent in a specified time interval.
Q15	Select join with Top-K	For users who joined the network in a given time interval, find their ids and the number of messages each sent in a specified time interval; report the top ten users with the most messages.
Q16	Spatial join	For each chirp message sent within a given time interval, find the ten nearest chirp messages.
U1	(Batch) Insert	Given the information for a set of new users, add the information to the database.
U2	(Batch) Delete	Given a set of existing user ids, remove the information for each user from the database.

TABLE I
BIGFUN OPERATIONS DESCRIPTION

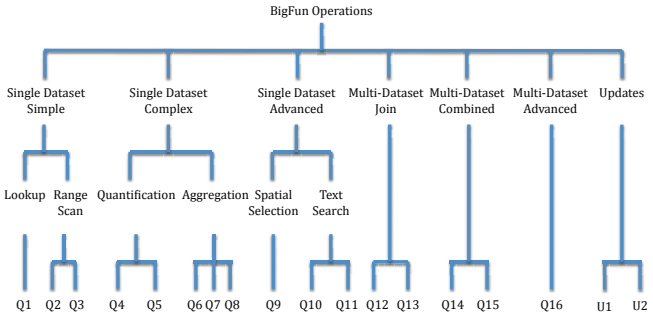

Fig. 3. BigFUN operations

of GleambookUsers. The aggregation queries (Q6, Q7, Q8) access the ChirpMessages dataset and aggregate over the results of the 'string-length' function. Q6 simply calculates this aggregate value for a subset of messages, Q7 adds grouping on top, and Q8 extends it with ranking and limit.

G3. Single Dataset - Advanced: This group evaluates fetching ChirpMessages' records using spatial or textual similarity predicates. Q9 uses *sender_location* to select messages sent from a given region whose boundaries are varied to create different versions of the query. Q10 and Q11 consider messages' contents. Q10 uses an exact string matching predicate, while Q11 uses edit distance as the similarity metric. The popularity of the search keyword changes in different query versions.

G4. Multi-Dataset - Join: This group considers joining GleambookUsers and GleambookMessages records. Q12 is a regular equi-join, and Q13 involves a left-outer join.

G5. Multi-Dataset - Combined: This group of queries (Q14 and Q15) measures the performance of a system for combinations of joins and aggregations. They use GleambookUsers and GleambookMessages to join, group, and rank a selected set of users based on their numbers of sent messages.

G6. Multi-dataset - Advanced: The last group of read-only queries considers a spatial join on the ChirpMessages dataset which returns the top-10 "near by" messages, ranked by sending time, for a set of Chirp messages. A "near by" message for a given message is one sent from a specified neighborhood (with varying boundaries in different query versions) around the sending location of the original message.

Updates: This group contains two data modification operations (U1 and U2) to study a system's behavior when dealing with data additions (insert) or removals (delete). A single insert (or delete) operation adds (or removes) a GleambookUser record and all its attribute values to (or from) the database. In the normalized schema case, a single operation turns into multiple corresponding operations on several datasets. As real applications do inserts and deletes both individually and in batches, BigFUN includes both singleton and bulk operations, with varying sizes.

V. EXPERIMENTS

This section presents a set of results obtained by running BigFUN on the systems in Section 3. We detail the setup and present the read-only and update results separately.

	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	Mongodb
9 Parts	169	285	292	18	276
18 Parts	338	571	587	36	553
27 Parts	508	856	881	53	833
Updates	162	279	272	-	278

TABLE II
TOTAL DATABASE SIZE (IN GB)

Dataset	Attribute	AsterixDB	MongoDB
GleambookUsers	user_since	BTree	BTree
GleambookMessages	author_id	BTree	BTree
ChirpMessages	send_time	BTree	BTree
ChirpMessages	sender_location	RTree	2d-IX
ChirpMessages	message_text	inverted-IX	text-IX

TABLE III
NESTED SCHEMA - SECONDARY INDEX STRUCTURES

Table	Column	System-X
GleambookUsers	id	BTree (PIX, Clustered)
GleambookUsers	user_since	BTree
GleambookMessages	message_id	BTree (PIX)
GleambookMessages	author_id	BTree (clustered)
ChirpMessages	chirpid	BTree (PIX, Clustered)
ChirpMessages	send_time	BTree
Employments	id	BTree (clustered)
FriendIds	id	BTree (clustered)
ReferredTopics	chirpid	BTree (clustered)

TABLE IV
NORMALIZED SCHEMA - INDEX STRUCTURES

A. Setup

We used a 10-node IBM x3650 cluster with a Gigabit Ethernet switch. Each node had one Intel Xeon E5520 2.26GHz CPU (4 cores), 12 GB of RAM, and four 300GB, 10K RPM hard disks, running 64-bit CentOS 6.6. On each machine three disks were used as data partitions to store persistent storage files. The fourth disk stored the transaction and systems' logs.
Client: The BigFUN client driver ran on a separate machine with 16 GB of RAM and 4 CPU cores connected to the same Ethernet switch. We used a single-user, closed-system model for our tests. As the performance metric, we measured the average end-to-end response time per operation (on a warmed up system cache) from the client's perspective.
System-X: For System-X, we used a version dated approximately 2013. Each node in the cluster served 3 database partitions. Data was partitioned using the system's hash partitioning scheme. We used the vendor's JDBC driver for the tests.
Hive: We used Apache Hive 0.13 with tables stored in the *Optimized Row Columnar* (ORC) file format. We configured 4 map and 4 reduce slots per node. Three disks per machine were used by HDFS to store data (with a replication factor of 1) and the fourth stored Hadoop logs.
MongoDB: We used MongoDB version 2.6.3. Our client used Java driver version 2.12.4. Collections were sharded using hashed keys. Each machine hosted three shards on separate disks. The fourth disk on each node stored the journal files.
AsterixDB: We used version 0.8.7 and its HTTP API. We used internal datasets, hash partitioned by primary key. One node controller with 6 GB of memory, 1 GB of bufferpool, and three data partitions ran on each machine. We measured AsterixDB's performance for two data type definition approaches for each data type: *AsterixSchema*, where we pre-declared all possible attributes, and *AsterixKeyOnly*, where we only defined the minimal set of attributes (required for indexing). These two variations lie on the end-points of the semi-structured data type definition spectrum in AsterixDB.

B. Read-Only Workload Results

Our read-only experiments focused on each system's scale-up using three scales with 9, 18 and 27 partitions on 3, 6 and 9 machines respectively. For the 9-partition scale, 90 million GleambookUsers, almost 450 million GleambookMessages, and more than 220 million ChirpMessages were generated. These cardinalities scaled up proportionally for the other two scales. We used the nested schema in Figure 1 for AsterixDB and MongoDB. For System-X and Hive, we used the normalized schema of Figure 2. Table II shows the total database size per system after loading. In each scale, the data size was at least five times the total available memory to make sure IO requests were not simply served by the OS cache. *AsterixSchema* and *AsterixKeyOnly* differed in size as the result of the extra information stored per record in the latter case. In Hive, the ORC file format's built-in compression reduced the total size of the stored tables significantly.
Auxiliary index structures: Table III lists the secondary indices created in AsterixDB and MongoDB for their nested schemas, while Table IV lists the indices used for the normalized schema in System-X (including table clustering). As an example, GleambookMessages were clustered on *author_id*, as queries tend to access them via this attribute to fetch all the messages for a given user. We gathered statistics in System-X to gain the information needed for good query plans. We did not use indices in Hive, as its optimizer does not automatically consider them; Hive users must manually reformulate queries so that indices can be used.

Lack of support for joins in MongoDB forced us to do this operation on the client side. Our client program performed an index nested loop join using the primary and secondary indices on the collections involved in the query.
Performance results: Table V shows the read-only queries' results. Each row shows the average response time (in seconds) for one query across all systems for all three scales. A cell with a '-' shows that we could not run a query against a system because the functionality or features tested by the query were not directly supported by that system. A cell with 'NS' means that the query failed to produce reliable results for that case. Each query may have several versions whose results are reported in separate rows. For most queries we have 3 versions: small (S), medium (M) and large (L), defined based on the selectivity of a query's filter and its semantics. In the base

	9-partitions					18-partitions					27-partitions				
	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB	Asterix *Schema*	Asterix *KeyOnly*	System-X	Hive	MongoDB
Q1	0.045	0.047	0.06	55.24	0.017	0.046	0.048	0.064	55.71	0.019	0.048	0.053	0.07	55.95	0.023
Q2-S	0.089	0.096	0.102	72.67	0.029	0.104	0.125	0.108	73.61	0.042	0.126	0.135	0.113	74.1	0.054
Q2-M	0.35	0.387	0.354	76.92	1.124	0.378	0.403	0.363	78.11	1.24	0.433	0.474	0.377	78.13	1.354
Q2-L	6.027	6.066	15.36	251.9	107.2	15.37	15.68	16.34	250.5	130.2	24.92	25.31	17.36	255.1	152.6
Q3-S	0.278	0.279	0.576	815.9	0.113	0.299	0.322	0.657	856	0.127	0.3	0.369	0.762	898.1	0.135
Q3-M	6.439	6.647	39.86	824.7	32.82	6.784	7.223	40.51	861	70.32	7.067	7.532	41.43	903.5	106.2
Q3-L	29.24	43.32	279.4	947.1	NS	32.04	47.48	294.1	1031	NS	43.58	63.81	312.8	1314	NS
Q4-S	0.28	0.307	0.335	526.6	0.133	0.316	0.336	0.34	545.3	0.138	0.319	0.359	0.345	570.5	0.141
Q4-M	6.503	6.603	15.66	533	49.07	6.715	6.881	15.87	554.2	117	6.732	7.491	16.27	575.9	183.3
Q4-L	31.56	44.31	76.85	583.8	NS	37.89	54.01	78.01	635.2	NS	57.81	80.72	80.13	686.9	NS
Q5-S	0.109	0.206	0.319	598.4	0.132	0.214	0.242	0.328	637.1	0.136	0.236	0.248	0.336	666.8	0.141
Q5-M	6.061	6.229	20.16	607.4	47.69	6.234	6.606	19.77	641	109.5	6.279	6.756	20.95	676.6	171.9
Q5-L	29.4	43.31	193.5	631.6	NS	33.52	51.92	194.6	648	NS	48.51	71.49	194.6	684.3	NS
Q6-S	0.246	0.247	0.126	51.29	0.174	0.25	0.268	0.135	50.91	0.19	0.271	0.281	0.148	52.07	0.208
Q6-M	6.769	6.806	4.581	51.82	9.057	6.78	6.865	4.776	52.37	10.13	6.872	6.886	4.924	52.94	11
Q6-L	76.51	112.5	90.98	53.86	NS	89.44	113.9	91.4	54.81	NS	93.99	116.5	93.32	56.02	NS
Q6-F	91.66	192.4	68.04	57.86	NS	92.59	200.3	70.12	58.8	NS	96.91	201	71.65	59.95	NS
Q7-S	0.468	0.475	0.133	54.55	0.199	0.553	0.612	0.136	55.81	0.204	0.712	0.740	0.136	56.41	0.212
Q7-M	6.975	7.022	4.739	57.49	9.987	7.082	7.304	4.901	58.11	10.61	7.411	7.687	4.925	59.92	11.3
Q7-L	96.11	200.6	89.43	66.1	NS	96.94	200.6	92.35	73.22	NS	100.1	200.7	94.69	80.54	NS
Q8-S	0.513	0.586	0.139	69.73	0.201	0.836	0.938	0.143	72.45	0.211	1.067	1.27	0.146	75.18	0.219
Q8-M	7.049	7.366	5.065	71.34	10.6	7.219	7.715	5.158	74.11	11.42	7.695	8.657	5.244	77.94	11.9
Q8-L	102.6	200.6	90.3	77.46	NS	147.63	200.7	93.11	78.98	NS	173.1	201	95.28	80.68	NS
Q8-F	200.8	323.8	135.9	151.1	NS	222.2	349.2	155.9	173.9	NS	236.1	386.8	176.9	196.3	NS
Q9-S	4.96	18.85	-	-	1.1	6.072	23.07	-	-	1.42	6.085	23.55	-	-	1.748
Q9-M	11.64	24.65	-	-	46.9	11.77	33.9	-	-	103.8	12.4	36.61	-	-	160.9
Q9-L	97.77	183.8	-	-	NS	102.9	193.3	-	-	NS	111.6	200.9	-	-	NS
Q10-S	94.54	174	-	-	990.7	95.82	175.7	-	-	1233	96.7	200.8	-	-	1481
Q10-M	100.2	190.4	-	-	1297	101.5	196	-	-	1393	103	198.3	-	-	1492
Q11-S	97.61	200.4	-	-	-	98.61	200.6	-	-	-	102	200.8	-	-	-
Q12-S	133.7	176.5	114	237.5	-	137.9	178.2	116.7	267.9	-	138.3	187.6	118.9	300	-
Q12-M	142.3	182.7	118.1	239.1	-	144.6	184.5	120	270.8	-	145.5	193.9	123.9	301.6	-
Q12-L	164.1	200.3	145	240.1	-	171.2	220.3	148.2	270.9	-	176.2	234	151.1	302	-
Q12-S-IX	1.078	1.091	1.219	-	1.149	1.703	1.823	1.918	-	2.769	2.289	2.524	2.552	-	4.396
Q12-M-IX	35.16	49.98	58.81	-	455.8	48.35	69.24	62.42	-	533.1	49.25	71.9	66.1	-	612.3
Q12-L-IX	120.9	192.1	142.4	-	NS	149.8	229.2	145.2	-	NS	155.7	241.3	146.9	-	NS
Q13-S	135.1	177.8	114.3	630.7	-	139	178.8	118.1	720.7	-	139.3	187.7	120.4	811.9	-
Q13-M	143.3	186.4	119.1	641.5	-	147.7	187.3	122.7	726.9	-	148.2	195.1	125.8	814.1	-
Q13-L	167.9	216.6	150.2	644.5	-	172.8	224.3	150.8	740.6	-	178.9	237.9	151.6	837.1	-
Q13-S-Ix	*	*	1.356	-	1.425	*	*	2.015	-	3.059	*	*	2.673	-	4.677
Q13-M-Ix	*	*	59.21	-	465.7	*	*	62.72	-	545.3	*	*	66.23	-	623.9
Q13-L-Ix	*	*	145.9	-	NS	*	*	146.7	-	NS	*	*	148.8	-	NS
Q14-S	140	184.6	114.2	293.5	-	144.5	188.7	117.1	296.9	-	144.8	189.5	118.9	299.7	-
Q14-M	148.5	193.8	119.2	296.6	-	153.2	197.4	120.5	298.8	-	154.5	199.7	122.6	302.1	-
Q14-L	166.6	213.4	151.3	297.5	-	169.3	222.6	152	301.1	-	173.3	235.1	151.5	304.5	-
Q14-S-Ix	1.142	1.174	1.423	-	-	1.714	1.834	2.051	-	-	2.293	2.709	2.677	-	-
Q14-M-Ix	37.51	52.61	59.16	-	-	50.42	71.48	62.81	-	-	51.51	72.14	66.48	-	-
Q14-L-Ix	121.1	193.6	142.8	-	-	152.7	235.4	145.9	-	-	157	243.7	147.4	-	-
Q15-S	140	184.8	114.5	313.7	-	144.6	189.8	117.1	315.7	-	144.9	189.8	119.2	319.2	-
Q15-M	148.8	193.9	120	315.3	-	153.3	198.2	123.2	319.7	-	154.7	200.9	125.5	323.1	-
Q15-L	169.5	216.1	151.3	316.7	-	170.6	224.6	151.4	320	-	174	235.3	151.7	323.3	-
Q15-S-Ix	1.25	1.479	1.467	-	-	1.81	1.941	2.161	-	-	2.475	2.757	2.842	-	-
Q15-M-Ix	38.01	53.39	59.73	-	-	51.91	73.87	62.97	-	-	53.3	74.32	67.57	-	-
Q15-L-Ix	121.8	195.6	143.6	-	-	152.8	238.7	145.3	-	-	157.2	244	148.8	-	-
Q16-S	1789	1876	-	-	21.89	1911	2014	-	-	57.41	2147	2236	-	-	90.19
Q16-M	1912	1996	-	-	68.41	2053	2207	-	-	148.1	2165	2482	-	-	230.7
Q16-L	4062	4315	-	-	NS	4662	4789	-	-	NS	4972	5052	-	-	NS

TABLE V
READ-ONLY QUERIES - AVERAGE RESPONSE TIME (IN SEC)

scale (9 partitions), the small filter selects an expected number of 100 records, while the medium and large versions select 10,000 and 1 million records respectively. These selectivites grow proportionally as the database scales up. Text search queries use keywords with varying popularities. The small and medium text filters use a keyword that occurs in at most 5% and 20% of the records.

In Table V, two of the aggregation queries, Q6 and Q8, also have a fourth version, denoted by F (standing for 'full'), in which the query runs over all records in the dataset.

Queries involving joins (Q12 to Q15) have two variations depending on the existence of an index on *author_id*. An 'ix' suffix in Table V denotes the indexed variation. The optimizer in System-X picks an indexed technique when it is expected to outperform other techniques. In AsterixDB, the user can add a hint to the query to change the default hybrid hash join technique to an indexed join. Currently, indexed join is only available for inner joins; For left-outer joins (cells with a '*' in Table V), a hybrid hash join is always used. For MongoDB we performed a client-side indexed nested loop join.

The results in Table V indicate that lack of automatic secondary index usage in Hive forced it to fully scan the involved dataset(s) in all the queries. This worsened its performance specifically for small queries. To be fair, Hive is designed for batch jobs over large datasets, not for short queries.

The "scatter-gather" nature of the queries made MongoDB's performance degrade when going from small to medium queries. For most of the large queries, we failed to obtain reliable results from MongoDB. We hit memory issues in running aggregation queries and observed time-outs for server-side cursors. MongoDB's Immortal cursors did not solve this issue, and they idled for long periods of time without returning results. MongoDB is known to behave well mostly for short queries that access one (or a few) shard(s).

C. Data Modification Workload Results

The data modification experiments were run against the GleambookUsers dataset (with an index on *user_since*) on 9 partitions. We tested batch sizes of 1 and 20 to see the impact of grouping insert or delete operations. Because of the nested schemas in AsterixDB and MongoDB, only one dataset and index needed to be updated per operation. System-X needed to update three tables and their indices due to normalization. We did not include Hive in these experiments as the life cycle of its data is maintained outside the system. Table II shows the initially loaded data size per system prior to the tests.

AsterixDB uses LSM-trees; updates modify the 'in memory' component of an index and they are appended to the transaction log for durability. We set the 'in memory' component budget to be 256MB per node in AsterixDB. In System-X, we increased the number and size of the transaction log files to improve performance. In MongoDB, we used the 'journaled' write concern to provide the same durability level as the other systems. We decreased the journaling commit interval to 2ms to make sure that our update client was not limited by the group commit policy.

	Batch Size	Asterix *Schema*	Asterix *KeyOnly*	System-X	Mongodb
U1	1	73.75	73.97	46.34	13.85
U1	20	6.20	6.23	30.15	7.9
U2	1	73.96	79.3	49.01	19.93
U2	20	4.73	4.89	33.79	14.2
Feeds	-	0.029	0.031	-	-

TABLE VI
DATA MODIFICATION OPERATIONS - AVERAGE RESPONSE TIME (IN MS)

We realized that these results will be meaningful only if the systems have warmed up enough. What is measured in the early part of an update workload is mostly the cost of in-memory updates. The warm-up phase must be long enough to fill the buffer cache with dirty pages prior to measurement.

Table VI shows the average response time (in ms) for 'one' data modification operation per system. For the batch size of 20, the average response time per batch is divided by the batch size. Grouping updates improved performance, as a portion of the overhead was amortized over multiple operations in a group. AsterixDB benefited the most from batching due to its current overhead for compiling and generating jobs per request. System-X did not benefit as much since it updated multiple structures; we also used JDBC prepared statements in System-X to reduce the overhead by letting the database run statements without having to compile them first.

Support for continuous data ingestion is a feature in AsterixDB that enables it to process fast streams of incoming data efficiently via a long running job. Table VI includes the performance of data feeds for the same insert workload, and the average insert time per record is several orders of magnitude faster than the U1 operation numbers.

VI. DISCUSSION

In this section, we summarize key lessons from this effort:
L1. Flexible schemas and their impact on performance: Many NoSQL systems support flexible schemas by storing heterogeneous records. They store enough information per record for later processing. This impacts queries that access a large number of records (such as Q6-L/F, Q7-L, Q8-L/F in our workload). Some systems, like MongoDB, choose maximum flexibility by being completely schema-less, while some like AsterixDB let users make a trade-off between flexibility and performance through data type definition. In our experiments, using the optional AsterixSchema data type definition allowed AsterixDB to achieve comparable performance to System-X.
L2. Pros and cons of normalized schemas: Normalization breaks a complex record into several parts and stores each in a separate dataset. Operations that include retrieving (or inserting) a large number of full records can suffer from a normalized schema as they access (and modify) several datasets and indices. Q3-L and U1 (with a batch size of 20) are examples where the performance of System-X was worse than the systems with nested schemas. However, queries that find all required attributes in one of the normalized tables can

benefit from normalization by not fetching unwanted data. Q6-F and Q7-L are examples where System-X skipped reading the *referred_topics* values per ChirpMessage record, unlike AsterixDB which had to read and discard this redundant data.

L3. Optimized storage format and its performance gain: The total size of the stored data impacts performance in queries with large scans. With the ORC format's reduced size and scan optimizations, Hive performed better than other systems in some queries with large scans such as Q6-F and Q7-L, where an aggregation was calculated on all the records.

L4. Advanced and mature query optimization: The query optimizer in a system plays a key role in exploiting index structures or run-time operators in a performant manner. For example, Q9-M in Table V shows that for spatial selections AsterixDB outperformed MongoDB using its spatial index. However, MongoDB showed much better performance for Q16-M (a spatial join using the same indices) by pushing down the limit clause into sorting and skipping unnecessary documents. AsterixDB failed to apply this optimization, although it had all the mechanisms for it. Another example is picking the indexed nested loop join versus hybrid hash join. System-X does that automatically, while AsterixDB requires a hint for this purpose. The mature, cost-based optimizer and advanced evaluation techniques in System-X enabled it to show stable performance in scaling up. At least for now this is an expected advantage for RDBMSs as compared to NoSQL solutions [16].

L5. MongoDB performance issues: MongoDB showed reasonable performance for short queries, but its performance dropped significantly for larger ones such as Q3, Q4, Q6, and Q7. The increased number of index lookups dropped the performance of the client-side join in larger versions of joins as well. (This performance could potentially be improved by techniques such as batching lookups.) As a NoSQL store, MongoDB targets a popular, but relatively narrow, set of use cases for performant processing.

L6. Job generation path and its overhead: The update results in Table VI show that the overall query path in AsterixDB currently add significant overhead for small operations. In our update tests, AsterixDB went from the slowest system to the fastest one with batching, as its job generation overhead was amortized over several operations. There are a number of known techniques such as parameterized queries, query plan caching, and simplified plan serialization, that AsterixDB still needs to add. Unlike AsterixDB, System-X (because of its mature optimizations) and MongoDB (because of its narrower operation scope) have built-in means to avoid this overhead.

L7. Performance changes when running updates: A system's performance can drop over time while running updates, especially in 'update in-place' systems such as MongoDB and System-X. This is due to increased page evictions in the buffer cache which increase the average response time in a system that has been serving requests for a while.

L8. "One size fits a bunch": The overall performance of AsterixDB shows it does not sacrifice core performance by delivering a wide range of functionality. If not the fastest, in most cases AsterixDB offered comparable performance to the fastest system. This supports its "one size fits a bunch" conjecture (which argues for a solid system with a rich set of features to serve different use cases rather than using multiple narrower systems), and it shows that it is apparently possible to build such a system.

VII. Conclusion

In this paper, we have reported on an evaluation of four representative Big Data systems using a micro-benchmark called BigFUN. We described the schema and operations in BigFUN and reported scaleup results for read-only workloads as well as the update performance of the systems. This work has attempted to look at Big Data benchmarking by evaluating supported features and their base performance in Big Data systems. We believe this direction in benchmarking is important, as we expect that Big Data systems will eventually converge on a set of features and operations to process both OLAP and OLTP workloads efficiently. Interesting future work could be expanding the set of systems, considering various distributions for data and query predicates, and adding multi-client tests that focus on the overall throughput of a system.

VIII. Acknowledgments

This work was supported by a UC Discovery grant, NSF IIS award 0910989, and CNS awards 1305430 and 1059436. We would also like to thank certain people affiliated with the System-X vendor for their valuable help on this work.

References

[1] B. Cooper et al., "Benchmarking Cloud Serving Systems with YCSB," in *SoCC*, 2010.
[2] A. Pavlo et al., "A Comparison of Approaches to Large-Scale Data Analysis," in *SIGMOD*, 2009.
[3] "MongoDB," http://www.mongodb.org/.
[4] "Apache Hive," http://hive.apache.org/.
[5] S. Alsubaiee et al., "AsterixDB: A Scalable, Open Source BDMS," *PVLDB*, 2014.
[6] D. J. DeWitt, "The Wisconsin Benchmark: Past, Present, and Future," in *The Benchmark Handbook*, 1991.
[7] O. Serlin, "The History of DebitCredit and the TPC."
[8] R. Cattell and J. Skeen, "Object Operations Benchmark," *ACM Trans. Database Syst.*, vol. 17, no. 1, 1992.
[9] M. Carey et al., "The OO7 Benchmark," in *SIGMOD*, 1993.
[10] M. J. Carey et al., "The BUCKY Object-Relational Benchmark," in *SIGMOD*, 1997.
[11] A. Schmidt et al., "XMark: A Benchmark for XML Data Management," in *VLDB*, 2002.
[12] A. Ghazal et al., "BigBench: Towards an Industry Standard Benchmark for Big Data Analytics," in *SIGMOD*, 2013.
[13] M. J. Carey, "BDMS Performance Evaluation: Practices, Pitfalls, and Possibilities," in *TPCTC*, 2012.
[14] S. Patil et al., "YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores," in *SoCC*, 2011.
[15] T. Rabl et al., "Solving Big Data Challenges for Enterprise Application Performance Management," *PVLDB*, 2012.
[16] A. Floratou et al., "Can the Elephants Handle the NoSQL Onslaught?" *PVLDB*, 2012.
[17] T. Armstrong et al., "Linkbench: A Database Benchmark Based on the Facebook Social Graph," in *SIGMOD*, 2013.
[18] "GridMix," https://hadoop.apache.org/docs/r1.2.1/gridmix.html.
[19] "PigMix," https://cwiki.apache.org/confluence/display/PIG/PigMix.
[20] S. Barahmand and S. Ghandeharizadeh, "BG: A Benchmark to Evaluate Interactive Social Networking Actions," in *CIDR*, 2013.
[21] P. Pirzadeh et al., "BigFUN: A Performance Study of Big Data Management System Functionality (extended version)," http://www.ics.uci.edu/~pouria/bigfun/BigFUN_extended.pdf, 2015.