SmartBench: A Benchmark For Data Management In Smart Spaces

Peeyush Gupta, Michael J. Carey, Sharad Mehrotra, Roberto Yus
University of California, Irvine
peeyushg@uci.edu, mjcarey@ics.uci.edu, sharad@ics.uci.edu, ryuspeir@uci.edu

ABSTRACT

This paper proposes SmartBench, a benchmark focusing on queries resulting from (near) real-time applications and longer-term analysis of IoT data. SmartBench, derived from a deployed smart building monitoring system, is comprised of: 1) An extensible schema that captures the fundamentals of an IoT smart space; 2) A set of representative queries focusing on analytical tasks; and 3) A data generation tool that generates large amounts of synthetic sensor and semantic data based on seed data collected from a real system. We present an evaluation of seven representative database systems and highlight some interesting findings that can be considered when deciding what database technologies to use under different types of IoT query workloads.

PVLDB Reference Format:
Peeyush Gupta, Michael J. Carey, Sharad Mehrotra, Roberto Yus. SmartBench: A Benchmark For Data Management In Smart Spaces. *PVLDB*, 13(11): 1807-1820, 2020.
DOI: https://doi.org/10.14778/3407790.3407791

1. INTRODUCTION

The emerging IoT revolution [14] promises to impact almost every aspect of modern society. In an IoT setting, sensors help create a fine-grained digital representation of the evolving physical world, which can be used to implement new functionalities and/or bring transformative improvements to existing systems. Central to IoT applications is the database management system (DBMS) technology that can represent, store, and query sensor data. Given the importance of IoT, a multitude of DBMSs, whether they be standard relational systems, key-value stores, document DBs, or specialized systems such as time series stores, have begun branding themselves as being suitable for IoT applications.

IoT systems pose special requirements on DBMSs. IoT data can be voluminous and is often generated at high speeds – a single smart phone contains dozens of sensors may generate data continuously every few seconds, a medium office building has several thousand HVAC sensors producing data at a similar rate, and a university/office campus may consist of several hundred thousand of such sensors. Sensor data is typically time-varying arrays of values and queries over sensor data involve temporal operators and aggregations. In a typical IoT setting, the underlying DBMS has to process observations captured from heterogeneous sensors (e.g., cameras, thermal sensors, GSR sensors, wearable technologies, etc.) each producing data with different structure.

Finally, the sensor data is often too low-level for being useful for the final applications directly and needs to be enriched/translated into higher-level semantic inferences. For instance, to provide personalized thermal comfort, an application may need to know the occupancy of different parts of a building, which may need to be inferred from diverse sensors including camera data, WiFi connectivity information, door sensors, and bluetooth beacons. Such interpretation is often performed using complex machine learning algorithms [35] outside the database system.

While different DBMSs provide different functionalities, none provide a comprehensive solution to the above challenges. Big Data management technologies built on top of cluster computing frameworks (*e.g.*, Hive [45], SparkSQL [19]), provide efficient ways to run complex OLAP queries on large amounts of data spanning multiple machines. However, they fail to do well on fast data ingestion, simple selections, and real-time queries. In contrast, stream processing systems like Apache Kafka [1], Storm [2], and Flink [22] provide faster response times on window-based continuous and real-time queries but do not perform well on historical data queries. Relational database systems, such as PostgreSQL [9], make use of indices and better join and aggregation operator implementations, but they do not scale well when the data becomes large and inherently do not support storage of heterogeneous data. Document stores like MongoDB [8], Couchbase [3], and AsterixDB [16] have an applicable logical data model and can easily store heterogeneous data, but do not provide special support for time-series data. Specialized time series database systems, such as InfluxDB [7] and GridDB [5], provide fast ingestion rates and fast selection on time ranges but they fail to support complex queries. IBM DB2 event store [6] is an in memory database that provides a very fast ingestion rate along with streaming data analysis capabilities.

The DB community has traditionally relied on benchmarks to understand the trade-offs between different DBMSs (e.g., the TPC-H [24] benchmark has been widely adopted by both industry and academia). In recent years, IoT DB benchmarks have begun to appear (e.g., [17, 18, 25, 34, 41, 42]) but the focus of such benchmarks has been on comparing systems based on fast ingestion of streaming sensor data. While ingestion is critical, IoT settings also require DBMSs

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407791

to support one-shot queries over IoT data both for real-time applications and for data analysis.

We present *SmartBench*[1], a benchmark focusing on evaluating DBMSs for their suitability in supporting both real-time and analysis queries in IoT settings. SmartBench, derived from a real-world smart building monitoring system (currently deployed in several smart buildings of the campus of the University of California, Irvine [12]) is comprised of an IoT data model, a set of representative queries, and a tool to generate synthetic IoT datasets. The schema captures the main concepts related to IoT environments and is, thus, extensible to different environments. It supports heterogeneity of sensors by using a semi-structured representation that can be mapped to the data model supported by the underlying DBMS. It also presents several mappings of such a representation to underlying DBMSs.

To provide a holistic view of the efficiency and capabilities of different DBMSs in supporting real-life IoT systems, SmartBench includes a mixed set of eleven representative queries that arise when transforming low-level sensor data into semantically meaningful observations (e.g., as a result of data enrichment during insertion), as a result of applications running on IoT data in real time, and during IoT data analysis. In addition, we include a mixed query workload that includes online insertions of sensor data as well as queries. Finally, SmartBench includes a tool to generate synthetic IoT datasets of different sizes based on real data used as a seed. The tool preserves the temporal and spatial correlations of the generated data, as this is important in order to evaluate systems in a realistic environment.

While SmartBench is based on a smart building context, it can be applied to other IoT systems as well because of its flexible schema which divides an IoT space into data, domain and device layers. We provide detailed performance results and an analysis of the performance of seven representative database systems (covering different technologies such as time series and specialized databases, relational DBs, document stores, and cluster computing frameworks). We test scalability (both scale up and scale out) of different DBMSs by comparing their performance under different data loads for a single node as well as a multi-node setup. From the results we have affirmed the intuition of a lack of a silver bullet. However, we have seen that some issues of specialized databases with respect to more traditional approaches (like the lack of support for complex operations) can be mitigated through external code that for typical IoT operations can perform adequately. We conclude with some interesting key observations that can help IoT system developers select DB technology(ies) for their data management needs and guide future DB developers to support such needs.

2. RELATED WORK

Widely used benchmarks, such as TPC-H [24] and TPC-DS [37], for analytical data processing and OLAP, and TPC-C [13] for OLTP, focus on traditional data management scenarios and do not address challenges posed by IoT environments. This has led to several new benchmarking efforts.

IoT benchmarks. The TPCx-IoT [41] benchmark was introduced for IoT gateway systems. The TPCx-IoT data model is based on modern electric power stations with more than 200 different sensor types. Its workload consists of concurrent inserts and simple range queries on recent data and does not include more complex queries (e.g., none involving joins). IoTABench [18] focuses on a smart meter use case and includes queries that focus on mainly on computations in such a scenario (e.g., bill generation). RIotBench [42] is based on real world IoT applications that aims at evaluating systems using streaming time-series data workload. Neither RIoTBench nor IoTABench consider the OLAP aspects of an IoT system focusing instead on data ingestion, streaming data, and continuous queries. Finally, Dayal et al. [25] presented a proposal for a big data benchmark, with an IoT industrial scenario as a motivation, which is similar in nature to ours. Their benchmark design includes representative queries for both streaming and historical data, ranging from simple range queries to complex queries for such industrial IoT scenarios. However, they did not provide an implementation of the benchmark and hence, there is no comparison of different database systems.

Big Data and streaming benchmarks. Some of the challenges of IoT data management are present in other related areas such as Big Data and streaming. Benchmarks measuring the performance of stream processing engines, like Linear Road (single node streaming) [17] and StreamBench (distributed streaming) [34], are focused on testing the performance of real-time data insertion/processing and queries. Nexmark [46] is a streaming benchmark that includes analytical queries on streaming data, but it is based on online auction data and does not capture queries run in an IoT system. Big Data benchmarks [27, 40, 47, 32] compare systems on their ability to process a large volume of heterogeneous data. BigBench [27] provides a semi-structured data model along with a synthetic data generator that mimics the variety, velocity, and volume aspects of big data systems. BigBench queries cover different categories of big data analytics from both business and technical perspectives. BigFUN [40] is based on a synthetic social network scenario with a semi-structured data model. HiBench [32] and BigDataBench [47] compare systems performance for analyzing semi-structured big data, including testing the systems' ability to efficiently process operators like sort, k-means clustering, and other machine learning algorithms in a map/reduce setting. YCSB [23] is a collection of micro-benchmarks with a workload containing various combinations of read/write operations (both random and sequential) and access distributions; it is representative of simple key-value store applications. It is mostly used for benchmarking distributed key/value storage systems. SmartBench, on the other hand, focuses on benchmarking systems under an IoT based system workload. Furthermore, along with simple reads and writes, SmartBench compares database systems on complex queries including joins and aggregations. IoT systems have to deal with the amalgamation of both the challenges of running analytical queries on historical data (big data benchmarks) and testing insertions and queries on recent data (streaming data benchmarks). This combination is not addressed by the previously described benchmarks.

3. SMARTBENCH BENCHMARK

We begin discussion of SmartBench by first highlighting the goals that guided our design. SmartBench is designed to explore data management needs of pervasive sensing environments, where a single smart space may house a large variety of sensors ranging from video cameras, microphones,

[1]See `http://github.com/ucisharadlab/benchmark` for SmartBench (code/data generation tool) and a longer version of this paper.

thermostats, beacons, and even WiFi Access Points (which can sense which devices are connected to them). Sensors may overlap in the type of physical phenomena they can sense E.g., one can determine the occupancy of a location using connectivity of devices to a WiFi access point. Occupancy can also be estimated based on temperature or power draw (e.g., number of power outlets connected to devices). It can also be estimated using the number of times a motion sensor in front of an entrance trips, or by using people counters. Each of these sensor modalities have their advantages and disadvantages from the perspective of applications.

Challenge 1: DB technology must provide mechanisms for the dynamic addition/deletion of new sensor types without requiring the system to stop. The system must support the ability to store and query data from sensors of newly added types. In designing our benchmark, we took into account such heterogeneity by including a general and extensible schema that enables generation of different types of sensors. The schema is mapped to the underlying DBMS based on the data model supported by the system.

In general, data captured by sensors is too low-level for applications. In the occupancy estimation example, sensor data needs to be enriched to generate the right semantic observations. For instance, if a camera is used for occupancy resolution, an image processing mechanism to count the number of people must be used in conjunction to determine occupancy. Occupancy might also require merging several sensor modalities using appropriate fusion algorithms.

Challenge 2: DB technology must provide efficient ways to support data enrichment. In general, data enrichment requires the use of specific functions that are executed frequently and can improve the efficiency of the task if executed directly in the database. For example, to compute the location of a person carrying a smartphone connected to the WiFi network, one would require a function such as sensor coverage which returns the geometric area that a sensor can observe. In our benchmark we have designed queries that encapsulate some of these typically required functionalities.

In smart environments, key concepts/entities are those of space and people immersed in the space, and most analytic applications revolve around discovering/analyzing relationships between people or between a person and the space. Such analysis can involve real-time data (e.g., current occupancy of the building) or historical data (e.g., average occupancy over weekends for the past 6 months), or both (e.g., a continuous query that checks when the current occupancy is higher than average over the past duration).

Challenge 3: DBMSs need to support a wide range of applications with different demands ranging from simple queries over recent data to fairly complex analysis of historical data. Also, such queries might involve querying raw sensor data as well as abstractions built on top of raw data. This can result in complex queries on the higher-level semi-structured data model, where typical DB operations like joins and aggregation can be desirable. In our benchmark we have included queries of this kind, including those required for both real-time and analytical applications.

3.1 SmartBench Schema

The schema used in SmartBench is based on the Observable Entity Relationship (OER) Model [36] which, in turn, is based on SensorML [21]. OER is an extensible model that allows incorporating new/heterogeneous types of sensors, observations, spaces, and users. SmartBench complete data model is specified in a longer version of this paper [10]. In the following, we highlight key entities and concepts in a smart space categorized into three interrelated layers.

Device Layer. Devices (aka, sensors and actuators) can, in general, be either physical or virtual. *Physical sensors* are real devices that capture real-world observations to produce raw data, whereas *Virtual sensors* are functions that take data from other sensors (physical or virtual) to generate higher-level semantic observations. A virtual sensor to detect the presence of people in the space, for instance, can use observations generated by physical sensors (e.g., images and connectivity of different MAC addresses to WiFi APs). Virtual sensors can be significantly more complex and may even include classification tasks based on machine learning models on past and streaming data.

Each sensor has attributes *type*, which dictates the type of observation the sensor generate, and *coverage*, which corresponds to the spatial region in which the sensor can capture observations. For physical sensors, coverage is modeled deterministically [49] and simplified as a function of its location - e.g., the coverage of a camera is its view frustum. For virtual sensors, coverage is a function of the coverage of the sensors used as their input - the exact function depends upon the specificity of virtual sensor. For example, the coverage of the presence detector (that determines location of people in the immersed space) based on camera inputs is the union of the view frustums of all the input cameras. Each physical sensor has a *location* and each virtual sensor has a property, *transformer code*, corresponding to the function applied to input sensor data to generate semantic observations.

Observation Layer. Sensors generate *observations* which are the units of data generated by a sensor. Physical sensors generate *raw observations*, whereas, virtual sensors generate *semantic observations* that correspond to a higher-level abstraction derived from raw observations. For example, a camera feed provides raw observations, whereas, the interpretation from the camera feed that a subject "John" is in "Room 2065" is a semantic observation. Services/applications are typically significantly easier to build using semantic observations (compared to raw observations) since one does not need to interpret/extract such observations from raw data repeatedly in application logic. Instead, such an abstraction is explicitly represented at the database layer.

All observations have a *timestamp* and a *payload*, which is the actual data (e.g., an image or an event). Semantic observations also have an associated *semantic entity*, which is the entity from the domain layer to which the semantic observation is related (e.g., a person or a space).

Domain Layer. This layer is comprised of the spatial extent of the smart space and information about subjects who inhabit it. Both of these concepts are inherently hierarchical, with the hierarchy representing granularity (e.g., a campus may consist of buildings, which have floors, which, in turn, are divided into rooms and corridors; likewise people are divided into groups – such as faculty, students, etc.). Domain entities have associated attributes which are classified as *static* or *dynamic*. These attributes, introduced in the W3C SSN ontology [15], model relevant and high-level information that smart applications would require about the space itself (e.g., its structure and functioning) or people within. Static attributes (e.g., the name of a room or a person, the type associated with rooms such as meeting room/office) are typically immutable, while dynamic attributes (e.g., the occupancy or temperature of a room, the location of a person) change with time. Dynamic attributes are *observable* if they can take (physical or virtual) sensor input to determine their value. Observable attributes are mapped to sensors using a function (*inverse coverage*) that given an entity (i.e., users/spaces), the type of observations required, and time, returns a list of sensors that can generate observations of the required type observing the required entity at the required time. For instance, *Inv_Coverage(Room 2065, t1, occupancy)* will return a set of (virtual) sensors that can output the occupancy of Room 2065 at time t1.

To implement the above model in a DBMS, one needs to map the appropriate concepts (viz., domain entities, sensors, observations, etc.), into database objects. These mappings are database dependent, as we will explain in Section 3.4.

3.2 Queries & Insert Operations

The benchmark consists of twelve representative queries motivated by the the need to support diverse applications as mentioned in Section 3. The first six queries are on raw sensor data (selected to support different building administration tasks, as well as queries needed by virtual sensors to generate semantically meaningful data). The next four queries are on higher-level semantic data (viz., on the presence of people in the space and occupancy of such spaces) and are chosen to represent different important functionalities provided by applications. The last two queries capture window-based operations and continuous queries. Almost all of the queries have a time range predicate specified, as would be expected of queries in the IoT domain. Below, we describe the queries and rationale behind their selection.

• **(Q1) Coverage($s \in Sensors$)**: returns the coverage of a given sensor s. Such queries are posed every time raw sensor data is transformed into semantic observations.

• **(Q2) InverseCoverage(L, τ)**, where L is a list of locations, and τ is a sensor type: lists all sensors that can generate observations of a given type τ that can cover the locations specified in L. Inverse coverage is computed every time we execute a query over semantic observations that have not been pre-computed from raw sensor data. Given the rate at which sensor data is generated and the number and complexity of domain specific enrichments, it may not be feasible to apply all enrichments at the time of data ingestion. In such a case, enrichments have to be computed on the fly, requiring inverse coverage queries to first determine the sensor feeds that need to be processed.

• **(Q3-Q4) Observations($S \subseteq Sensors$, t_b, t_e)**: selects observations from sensors in the list of sensors S during the time range $[t_b, t_e]$. We differentiate between two instantiations of this query. Observation queries with a single sensor in S are referred to as Q_3. Such queries arise when applications need to create real-time awareness based on raw sensor data (e.g., continuous monitoring of the temperature of a region). Observation queries when S contains several distinct sensors (often of different types) are referred to as Q_4. Q_4 arises for a very different reason – as a result of merging several sensor values (using transformation code) to generate higher-level observations. Since the two types of queries arise for different reasons, and (as we will see) their performance depends upon how we map data into the database, we consider the two queries separately.

• **(Q5) C_Observations(τ, *cond*, t_b, t_e)**: selects observations generated by sensors of given type τ in the time range $[t_b, t_e]$ that satisfy the condition *cond*, where *cond* is of the form $\langle attr \rangle\ \theta\ \langle value \rangle$, *attr* is a sensor payload attribute, *value* is in its range, and θ is a comparison operator, e.g, equality. Such queries often arise in large-scale monitoring applications of multiple regions, e.g., monitoring spaces for abnormal (too high/low) temperatures.

• **(Q6) Statistics($S \subseteq Sensors$, A, F, t_b, t_e)**: retrieves statistics (e.g., average) based on functions specified in F during the time range $[t_b, t_e]$. The statistics are generated by first grouping the data by sensor, and further by the value of the attributes in the list A. For instance, a query might group observations on sensor_id and day (which is a function applied to timestamp) and compute statistics such as average. Such a query is important to build applications that provide the building administrator information about the status of sensors (e.g., if the sensors are generating too much data or none at all, discovery of faulty sensors).

• **(Q7) Trajectories(t_b,t_e, l_b, l_e)**: retrieves the names of users who went from location l_b to location l_e during the time interval $[t_b, t_e]$. Such queries arise in tasks related to optimizing building usage, e.g., for efficient HVAC control, janitorial service planning, graduate student tracking, etc.

• **(Q8) CoLocate($u \in Users$, t_b, t_e)**: retrieves all users who were in the same Location as user u during the specified time period. Any application involving who comes in contact with who in which location (e.g., to construct spatio-temporal social networks) runs such a query on the historical presence data of users.

• **(Q9) TimeSpent($u \in Users$, η, t_b, t_e)**: retrieves the average time spent per day by subject u in locations of type η, (e.g., meeting rooms, classrooms, office, etc.) during the specified time period. This query arises in applications that provide users with insight into how they spend their time on an average during the specified period.

• **(Q10) Occupancy(L, Δ_t, t_b, t_e)**: retrieves the occupancy level for each location in the list L every Δ_t units of time within the time range $[t_b, t_e]$). This query is the direct result of a requirement to visualize graphs that plot occupancy as a function of time for different rooms/areas.

• **(Q11) Occupancy Smoothing(L, t_b, t_e)**: retrieves the smoothed out occupancy levels for each location in the list L, within the time range $[t_b, t_e]$). The smoothing is done by taking average of last 10 occupancy level after subtracting minimum and maximum occupancy level out of the 10 readings. This query produces a smooth and easy to follow occupancy graph to the end-users.

• **(CQ) Continuous Query(η, α, β)**: retrieves, after every α seconds (hop size), the minimum, maximum, and average occupancy levels of locations of type η in the last β seconds (window size). This is a continuous query which is executed as the data is being ingested into the database.

• **(I) Insert($s \in Sensors$, t, *payload*)**: inserts a sensor observation at time t into the database.

• **(IE) Insert&Enrich($s \in Sensors$, t, *payload*, *params*)**: takes as input (raw) sensor observations and mimics execution of an enrichment pipeline to generate and insert semantic observations. Such a pipeline typically consist of a sequence of operations. Based on the type of data, it first identifies the set of enrichment functions that need to be invoked. Each of those functions, may result in one or more queries over both metadata or data (e.g., queries of types Q1-Q5). For instance, to transform a WiFi connectivity event (indicating that a given device connected to a specific AP at a given time) a query of type Q1 may be executed to determine location of the AP following which sensor data from neighboring APs about the device (or other devices) might be accessed (queries of type Q3-Q5) to predict the semantic location (e.g., the room) in which the owner of the device might be. Likewise, for a camera sensor, image segmentation and face recognition may need to be performed to determine the identity of a person in the view. Since enrichment functions (e.g., face recognition, location determination, etc.) are performed outside of the DBMSs [48], the choice of specific enrichment function does not influence suitability of a DB technology to IoT application. We simply model the I&E pipeline as a sequence of queries followed by a busy wait (to mimic an execution of the enrichment function), followed by an insertion of the semantic observation. The specific queries generated in the pipeline and the wait time are *parameters* which are input to the IE function.

The set of queries listed above represents an important functional aspect of building smart space applications that have been motivated by the campus-level smart environment that we have built at UCI. Note that these queries represent sample IoT data management tasks involving operations including selections, joins, aggregations, and sorting.

3.3 Data And Query Generator

Benchmarks typically offer tools to generate datasets of varying size to test systems in different situations [29, 43, 31, 44, 30, 18]. SmartBench's data generation tool[2] uses seed data from a real IoT deployment (our University building) including sensor data and metadata about the building and sensors, which is able to scale up/down to create a synthetic dataset. The tool can scale the IoT space (i.e., number of rooms, people, sensors) as well as sensor data (i.e., number of observations per second, time period during which the sensors produce data, etc.). The tool preserves temporal and spatial correlations in the seed data to support realistic selection and join queries. We developed our data generation tool, rather than using a uniform distribution for every attribute independently, since we aimed at comparing different DBMSs with more realistic data (for a smart space setting). The data patterns (e.g., variation of the occupancy values between the day time and the night time, variation in the number of WiFi connections at different points of time) affect the execution time of queries based on the query parameters. To maintain a fair comparison, we execute exactly the same instances of queries on the same data for each DBMS. We do not aim at characterizing types of smart space data based on the patterns it contains.

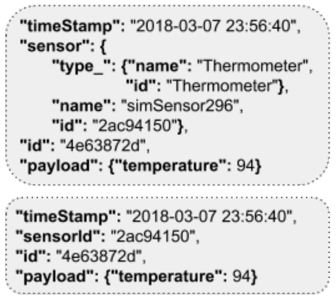
In addition, the tool also generates an actual query workload based on the query templates described above. Its input is the already generated metadata and a configuration file containing different parameters. All of the queries except Q1 and Q2 include a time range based filter $[t_b, t_e]$. t_b is selected at random from the range $[T_b, T_e]$ where T_b and T_e are the minimum and maximum timestamp of the entire observation dataset, respectively, and t_e is selected at random from the range $[t_b + \delta_a, t_b + \delta_b]$ where values for δ_a and δ_b are provided in the configuration file. List based query parameters, e.g., the list of sensors in query Q4 and Q6 and the list of locations in query Q10, are generated by randomly picking (without replacement) n elements from the already generated metadata; n itself is selected at random from the range $[n_a, n_b]$, where n_a and n_b are provided in the configuration file. Scalar parameters like user u in query Q8 are selected at random from the available values in the metadata.

[2]An extended explanation of the tool, including configuration parameters supported, is available in [10].

3.4 Model Mapping

There are several ways in which the schema in Section 3.1 can be represented in the underlying DBMS. We explore multiple mappings of IoT data to the underlying databases, which are broadly characterized based on how sensor data is stored in the database: *(A)* Single table for observations from all sensors; *(T)* Multiple tables with one per sensor type; and *(S)* Multiple tables with one per sensor instance.

A given mapping approach can be simple and reasonable to apply on some database systems, while difficult or unnatural for others. With PostgreSQL, mapping T is straightforward, while mapping A can be applied using its JSON data type (although not intuitively since the JSON type is stored as a BLOB). Mapping S is not practical since it would create a very large number of tables. Since CrateDB support structured data, mapping T is most natural while A can again be applied using a JSON data type since CrateDB internally store data in the form of documents. For document stores (e.g., AsterixDB and MongoDB), mapping A is the most natural due to their support for a semi-structured data model). We generate two distinct strategies for document stores (see Figure 1 for examples of both) – a *Sub-Document Model (A1)*, where each observation is stored as a nested document by embedding related data together (however, since complete nesting can create very large documents, we only nested those attributes that can help in reducing the number of joins), and a second *Normalized Model (A2)* where each observation is stored as a fully normalized document, where every relationship is modeled as a reference between two documents.



```
"timeStamp": "2018-03-07 23:56:40",
"sensor": {
    "type_": {"name": "Thermometer",
              "id": "Thermometer"},
    "name": "simSensor296",
    "id": "2ac94150"},
"id": "4e63872d",
"payload": {"temperature": 94}
```

```
"timeStamp": "2018-03-07 23:56:40",
"sensorId": "2ac94150",
"id": "4e63872d",
"payload": {"temperature": 94}
```

Figure 1: Mappings for document stores (A1 and A2).

Timeseries databases such as InfluxDB and GridDB do not support JSON and hence cannot support mapping A directly. However, as mentioned in Section 4, GridDB provides two types of fixed schema containers, general purpose and time series only (time stamp as primary key). We can use the general purpose containers to store observations/semantic observations from each type together in one container (though, a container in GridDB can not be partitioned and therefore this model will not scale well for GridDB). Also, for GridDB, mapping S can be realized using its specialized time series container for storing sensor data, where observations/semantic observations from a single sensor can be stored in a separate time series container. InfluxDB stores time series data in terms of *data points.* A point is composed of: a *measurement*, a list of *tags* (each tag is a key value pair used for storing metadata about the point), a list of *fields* (each field is a key value pair use for storing recorded scalar values), and a timestamp. Tags are indexed in InfluxDB but fields are not. Related points having different tag lists but generating the same types of readings can be associated into a measurement (synonymous with a SQL table). We create separate measurements for different sensor types and store Observations/Semantic Ob-

Table 1: Different DBMS and their capabilities.

DBMS	Secondary Indexes	Joins	Aggregation	Column/ Row Store	Structured/ Semi-Structured	Compression	Storage Structure	Query Language
PostgreSQL	Yes	Yes	Yes	Row	Structured[a]	No	In-place update[b]	SQL
AsterixDB	Yes	Yes	Yes	Row	Semi-Structured	No[i]	LSM	SQL++
MongoDB	Yes	Yes[c]	Yes	Row	Semi-Structured	Block	In-place update[b]	MongoDB QL
GridDB	Yes	No	No	Row	Structured	Block	In-place update[b]	TQL[d]
CrateDB	Yes[e]	Yes	Yes	Column	Structured[a]	No	Inverted Index	SQL
SparkSQL	No	Yes	Yes	Column	Structured[a]	Columnar[h]	Dataframes	SQL
InfluxDB	No[f]	No	Yes	Column	Structured	Columnar	TSM[g]	InfluxQL[d]

[a]provides a JSON column type to store semi-structured data, [b]heap files supported by BTree indexes, [c]only with an unsharded collection, [d]subset of SQL, [e]requires index on every column used in where clause, [f]tags are implicitly indexed but the values cannot be indexed, [g]TSM (time structured merge trees) are similar to LSM trees and store data in read-only memory-mapped files similar to SSTables, however these files represents block of times and compactions merge blocks to create larger blocks of time, [h] Parquet files on HDFS, [i]AsterixDB recently added support for compression that will be generally available in its next release.

servations of different types in different measurements. InfluxDB does not provide any means of storing non-time series data, so we cannot apply any mapping other than mapping T and we cannot store all of the metadata. Hence, we store the building metadata in a PostgreSQL database when using InfluxDB. This metadata is fetched from PostgreSQL database whenever it is required by the application.

4. DATABASE SYSTEMS

To evaluate a wide range of database technologies in supporting analytic workloads on IoT data, we broadly classify systems into four categories: traditional relational database systems, timeseries databases, document stores, and cluster computing based systems. For our experiments, we selected representatives from each category to cover a wide range of data models, query languages, storage engines, indexing strategies, and computation engines (see Table 1). Our main considerations in selecting a particular DBMS were that it: (a) Provides a community edition widely used for managing and analyzing IoT data at many institutions; (b) Is popular based on its appearance on the DBEngine website [4]; or (c) Is advertised as specialized timeseries database system optimized for IoT data management.

In our selection, we did not restrict systems based on their specific underlying data model or their query language. We appropriately convert our high-level data model and corresponding queries to the data model and query language supported by the database system. In IoT use cases, the sensor data is typically append only and updates are applied only to the metadata. We, therefore, require database systems to support atomicity at the level of single row/document but do not require or use multi-statement transactions. For the systems that provide stricter transaction consistencies, we set the weakest consistency level that provides atomic single row/document insert so that database locking would not affect the performance of the inserts and queries.

Relational database systems. From this category, we chose *PostgreSQL* since it is open source, robust, and ANSI SQL compatible. PostgreSQL supports a cost-based optimizer and also supports a JSON data type, which is useful in storing heterogeneous sensor data. PostgreSQL has a large user base with many deployments for IoT databases.

Timeseries database systems These systems are optimized to support time varying data, often generated by machines, including sensor data, logs, events, clickstreams, etc. Such data is often append-only, and timeseries databases are designed to support fast ingestion rates. Furthermore, time series databases support fast response times for queries involving time-based summarization, large range scans, and aggregation over both real-time or historical data. We selected two time series database systems in our benchmark study: (a) *InfluxDB*, which is the most popular timeseries database at present (according to the DBEngine ranking [4]). Along with the typical requirements for handling time series data, InfluxDB provides support for various built-in functions that can be called on time series data. It has support for retention policies to decide how data is down sampled or deleted. It also supports continuous queries that run periodically, computing target measurements. (b) *GridDB*, which is a specialized time series database. We selected GridDB because it has a unique key-container data model. The data in the container has a fixed schema on which B-tree based secondary indexes can be created. The container is synonymous with a table in relational database system on which limited SQL functionality is available. GridDB, however, does not provide support for queries that involve more that one container, disallowing aggregation over more than one time series. Containers can be either general purpose or time series only containers. Time series containers have a time stamp as the primary key and provide several functions to support time-based aggregation and analysis.

Document stores. We selected two document stores as representatives of this category for our evaluation: (a) *MongoDB*, which is one of the most popular open source document stores that supports flexible JSON-like documents that can directly be mapped to objects in applications. MongoDB provides support for queries, aggregation, and indexing. It has sharding and replication built in to provide high availability and horizontal scaling. MongoDB supports multiple storage engines. In this study, we used MongoDB with WiredTiger, which is the default storage engine and supports B-tree indexes and compression (including prefix compression for indexes). (b) *AsterixDB*, which, much like MongoDB, stores JSON-like documents. It, however, supports many more features including a powerful semi-structured query language SQL++[38] (which is similar to SQL but works for semi structured data). AsterixDB is designed for cluster deployment and supports joins and aggregation operations on partitioned data through a runtime query execution engine that does partitioned-parallel execution of query plans. AsterixDB has LSM-based data storage for fast ingestion of data and it supports B-tree, R-tree, and inverted keyword based secondary indexes. It can also be used for querying and indexing external data (e.g., in HDFS).

Cluster computing frameworks like Hadoop and Spark provide distributed storage and processing of big datasets. Database query layers like Hive & SparkSQL, built on top, provide SQL abstraction to applications to increase portability of analytics applications (usually built using SQL) to cluster computing environments. We selected *SparkSQL* as a representative of this group. SparkSQL uses columnar storage and code generation to make queries fast. Since it is built on top of the Spark core engine, it can also scale to thousands of nodes and can run very long queries with high fault tolerance. These features are intended to make the system perform well for queries running on large volumes of historical data (business intelligence, data lakes).

Table 2: Summary table with pros and cons of different database technologies.

DBMS Technology	Pros	Cons	Systems	Impacts
Semi-structured data model	Flexible schema	No standard query language	AsterixDB, MongoDB	Mapping
Structured Model	Well established with standard query language (SQL)	Difficult to model complex and dynamic data	PostgreSQL, CrateDB, SparkSQL, GridDB, InfluxDB	Mapping
Full SQL or similar language support	No need to implement application level operators	None	PostgreSQL, CrateDB, SparkSQL, AsterixDB	Ease of use
Row Storage	Faster Inserts	Slower OLAP queries	PostgreSQL, AsterixDB, MongoDB, GridDB	I
Columnar Storage	Faster OLAP queries	Slower Inserts	CrateDB, SparkSQL, InfluxDB	Q6-Q10
LSM/TSM Trees	Faster writes	Slower reads	AsterixDB, InfluxDB, SparkSQL	I
Secondary Indexes	Faster reads	Slower writes	PostgreSQL, MongoDB, AsterixDB, GridDB, InfluxDB, CrateDB	Q1-Q10
Specialized Timeseries Databases	Fast inserts, fast selection and other simple queries	Limited functionality (no support for JOIN)	GridDB, InfluxDB	Q3-Q5
Sharding And Distributed Query Engine	Important for scaling out	None	AsterixDB, MongoDB, CrateDB, SparkSQL, InfluxDB	Scale Out

Other database systems. We also chose *CrateDB*, which is advertised as a SQL database for timeseries and IoT applications but does not fit the criteria for a specialized time series database system. CrateDB supports a relational data model for application developers but internally stores data in the form of documents supporting JSON as one of the data types. However, along with storing documents as is, CrateDB stores data in columnar format as well. Its distributed query engine provides full SQL support along with other time based functions. CrateDB is built on top of ElasticSearch [28] and therefore naturally supports inverted indexes, although it has no support for B-tree based indices.

Table 2 lists our apriori expectations regarding the impact of the DB technology choice on the performance of different queries based on factors such as row versus column based storage, support for LSM/TSM trees, temporal predicates, indexing mechanisms, query optimization and processing. The last column in the table lists our expectation on how each technology impacts different aspects of the benchmark. For instance, the choice of underlying data model (semi-structured/structured) would affect how SmartBench data model is mapped to the underlying data model supported by the database system, and the usability of the system.

5. EXPERIMENTS AND RESULTS

Dataset and queries. We used the data generator tool with seed data from a real IoT data management system, TIPPERS [12], deployed in the DBH building at UC Irvine. DBH is equipped with various kinds of sensors including 116 HVAC data points (e.g., vents and chillers), 216 thermometers, 40 surveillance cameras, 64 WiFi APs, 200 beacons; there are also 50 outlet meters that measure the energy usage of members of the ISG research group. The TIPPERS instance has been running for two years and has collected over 200 million observations from these sensors. This data also includes higher-level semantic information generated through virtual sensors about the presence of people within the building and occupancy levels.

We specifically used as seed TIPPERS data for one week from 340 rooms, observations from three types of sensors – 64 WiFi Access Points, 20 plug meters, and 80 thermometers– and semantic observations about location of people and occupancy of rooms. With this seed we generated three datasets with different sizes (see Table 3). The query parameters were also generated using SmartBench's generation tool. We controlled the query selectivity by restricting the time range to be $1\ day < t_e - t_b < 4\ days$ for the base experiments (larger periods of time were used in an experiment to test the impact of query selectivity on the systems' performance). The complete list of the query generation parameters used can be found in the extended version at [10]. For each benchmark query template, we generated 25 query instances with different parameters. We ran each generated query instance on every DBMS sequentially and their average execution time along with the variance is reported.

Database System Configuration. We used MongoDB CE v3.4.9, GridDB SE v3.0.1, AsterixDB v0.9.4, PostgreSQL v11.2, CrateDB v3.2.3, InfluxDB v1.7 and SparkSQL v2.4.0. For the client side, we used Java connectors for MongoDB and GridDB, the HTTP APIs for AsterixDB and InfluxDB, and JDBC connectors for PostgreSQL, SparkSQL, and CrateDB. We used default settings for most of the configuration parameters setting the buffer cache size to 2 GB per node for all. We created a secondary index on the timestamp attribute for all systems except for SparkSQL (since it does not support secondary indices). For CrateDB, we created indices on all columns since it requires an index on all columns that could be part of any selection predicate. For GridDB, with its container per sensor model (S), we created a primary index on timestamp, since the timestamp needs to be a primary key in GridDB's time series containers.

5.1 Single Node Experiments

We tested the DBMSs using the datasets in Table 3 on a single node (Intel i5 CPU 8GB RAM, 64 bit Ubuntu 16.4, and 500 GB HDD).

Experiment 1 (Insert Performance - Table 4): We compare the insert performance on a hot system (with 90% data preloaded). We used single inserts for GridDB and InfluxDB, batched prepared statements for PostgreSQL, CrateDB, and SparkSQL, batched document inserts for MongoDB, and socket based data feeds for AsterixDB to insert the 10% insert test data. The batch size used is 5,000 rows/-documents which amounts to 5 minutes worth of observation data for the large dataset. WAL is enabled for all the systems and the default configuration is used for compression in the systems that support it. We did not perform insert performance tests on SparkSQL since it does not manage storage by itself (instead it depends on external sources – Parquet files on HDFS in our case). Table 4 shows the size of the dataset after ingestion and ingestion throughput.

InfluxDB performs best due to its TSM based storage engine and to its support for columnar compression, which reduces the size of data inserted to about one-third. GridDB (with mapping S) performs the second best due to the following: 1) It maintains a relatively small size of data (compared to other row stores); 2) It flushes data to disk only when the memory is full (or due to checkpoints, the default value for which is 20 minutes), which achieves benefits similar to LSM storage; and 3) It does not flush WAL at every insertion, but periodically every 1 second[3]. With mapping T, GridDB still remains efficient – though it takes 25% more time compared to mapping S due to higher index update overheads.

[3]May result in a loss of the last 1 second of data in case of failure.

Table 3: Dataset sizes.

Dataset	Single Node		Multi Node	
	Sml	Lrg	Sml	Lrg
Users	1,000	2,500	10,000	25,000
Sensors	300	600	3,000	15,000
Rooms	300	600	2,500	7,500
Days	90	120	150	180
Frequency	1/300s	1/200s	1/150s	1/100s
Observations	14 mil	30 mil	150 mil	800 mil
S. Observations	14 mil	30 mil	150 mil	800 mil
Size	12GB	25GB	125GB	600GB

AsterixDB performs better than MongoDB due to its support for LSM storage even though MongoDB has a slightly smaller database size from insertions due to its row-level compression. PostgreSQL performs better than AsterixDB, even though it lacks LSM storage, due to the following: 1) The overall size of records stored in PosgreSQL is smaller compared to AsterixDB (which is a document store), thereby saving I/O; 2) PostgreSQL supports heap storage and new records are inserted in memory with the data spilling over to disk only when the memory is full. Such a storage mitigates many of the advantages of LSM for insert-only workloads, while preventing the processing overhead due to merging incurred by LSM storage; 3) In PostgreSQL, updates to index pages could result in random I/O, in contrast to AsterixDB (since its indexes are also LSM trees). However, the indexes created on primary key and time were relatively small (and mostly memory resident), thereby limiting the advantages of an LSM tree for index updates. CrateDB performed the worst given its columnar storage engine with no compression (by default). The requirement to create indexes on all the columns used for selections caused more index updates at insertion time. Data ingestion takes about 20% more time on mapping T compared to mapping A for CrateDB, since this mapping requires more columns to be indexed.

We conducted an additional experiment on insert with enrichment. The IE pipeline (see Section 3.2 consists of query Q1 for the sensor specified in IE operator (metadata query), followed by one of Q3 or Q4 (queries on the sensor data already ingested) chosen randomly. The parameters for Q3/Q4 are generated using the method mentioned in Section 3.3, with min and max time interval of 4 and 8 hours, respectively. We chose τ (busy wait to simulate execution of enrichment function) to be 100ms, based on the time taken for enrichment functions in [48] in the context of tweet processing. Sensor data enrichment, e.g., ML functions on images, could even take longer. Even for this relatively modest value of τ, enrichment cost quickly dominates and becomes a bottleneck and the rate at which semantic observations are generated (10,000/second) could easily be sustained by all the systems. Thus, the experiment did not further provide insight from the perspective of comparing across different DB technologies. Nonetheless, the result points to two interesting asides: (a) optimizing enrichment at the time of ingestion in DBs [48] is an important challenge to support real-time applications that need enrichment, and (b) scaling systems requires additional hardware where enrichment can be performed prior to data being stored in the DBMS.

Experiment 2 (Query Performance - Figure 4): We compare the performance for the benchmark queries in Section 3.2. For all systems, except GridDB and InfluxDB, every benchmark query maps to a single query in the query language supported by the system. For GridDB and InfluxDB, the benchmark queries that involve joins and aggregation cannot be directly executed due to the lack of support for such operations. Hence, we implemented these operators at the application level by pushing selections down, using the best join order, and performing the equivalent to an index nested loop based join. Similarly, for database sys-

Table 4: Dataset sizes after ingestion (including indices) and ingestion throughput for single node and multi node.

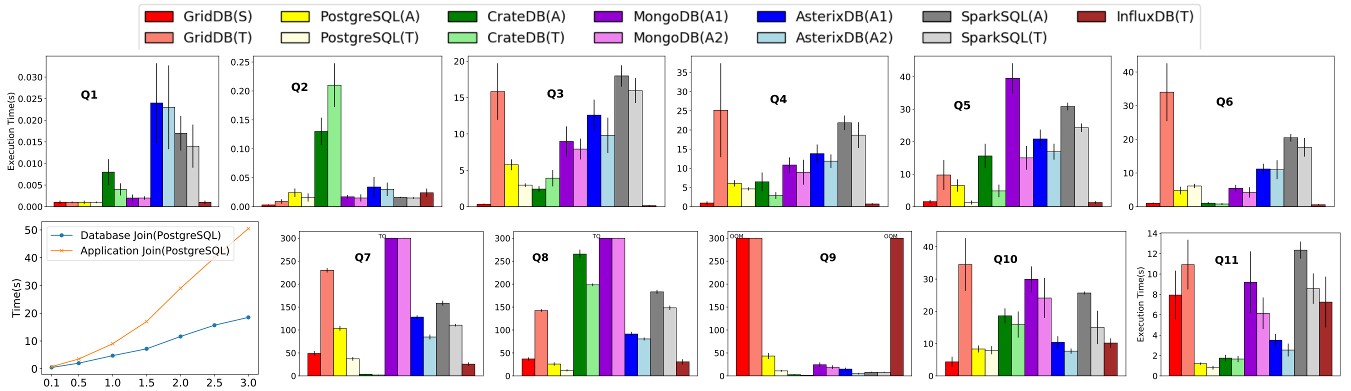
Dataset	Map	Data Size (GB)			Inserts/sec		
		Sml	Lrg	Multi	Sml	Lrg	Multi
griddb	S	4	8.6	85	42,425	32,500	**9,332**
	T	5.6	11		32,941	24,045	
postgresql	A	8	18	-	1,451	1,018	-
	T	7.5	17		5,490	5,110	
mongodb	A1	7	16	115	2,060	1,640	955
	A2	6.2	15		4,087	3,286	
asterixdb	A1	9	22	128	2,023	2,019	3,750
	A2	7.5	20		4,160	4,050	
cratedb	A	10	25	140	2,017	1,495	748
	T	12	30		1,565	1,138	
sparksql	A	7.5	14	95	-	-	-
	T	6.5	12				
influxdb	S	2.8	6.4	-	**59,222**	**58,320**	-

tems which do not support window queries (i.e., GridDB, InfluxDB, and MongoDB), we implemented an application-level window operator that first fetches data according to the where clause, partitions it based on the grouping key using in-memory hash table, and then sorts the list corresponding to each key based on the ordering attribute. Also, since InfluxDB does not support any way to store non-timeseries data, we stored the building metadata information (users, building info, sensor attributes) in PostgreSQL. The full version of the paper [10] shows the complete implementations of the queries for GridDB, InfluxDB, and the other database systems. Figure 4 shows the average execution time per query on the large dataset, along with standard deviation. Since the same query is run with 25 different sets of parameters (the parameters are the same across DBMSs), we see a significant variance in most queries. The longer the query execution time, the higher the variance, but variance to execution time ratio is larger for queries running within a second. Even with the observed variance, the relative performance among most of the systems can be compared.

Metadata Queries (Q1,Q2). All the systems performed relatively well on the metadata queries, included to compare the ability to store arbitrary data, except for InfluxDB which does not provide a way to store complex metadata.

Simple Selection and Roll Up (Q3-Q6). Time series DBs performed well on these queries, specially on Q3-Q5 which are range selection queries over timestamp: GridDB (mapping S) performs very well, since it stores data clustered based on timestamp (the primary key), and InfluxDB's performance is comparable (slightly better). PostgreSQL and CrateDB perform similarly since these queries required most of the columns to be retrieved and do not include any aggregation operations (the columnar storage of CrateDB did not provide much benefit). MongoDB and AsterixDB are slower since these queries involve scanning a set of rows and, due to the document model, they have to deal with larger record sizes. Also, AsterixDB's LSM tree based storage can slow down reads since the system has to first look for the corresponding primary keys in possibly multiple index files (due to LSMified secondary indexes) and then search for the corresponding rows in multiple LSM files (if the immutable files are not already merged). MongoDB performed slightly better than AsterixDB thanks to its WiredTiger storage engine supporting data compression and the use of index compression by default that helps in better secondary index scans.

Complex Queries (Q7-Q10). These queries involve complex joins and aggregations that are not natively supported in GridDB or InfluxDB. Hence, their processing requires the application-level implementation of these operators. At low selectivity (date range of only 1 to 4 days), the number of rows to be joined and grouped after all the execution of selection predicates is small and so application-level joins did not cause significant overhead. In fact, except for Q7 and Q8, GridDB and InfluxDB outperformed all the other databases (with native JOIN support) except for



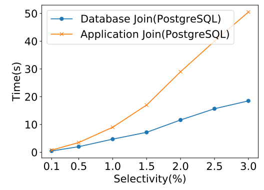


Figure 3: Performance of application vs DB joins.

*GridDB + application-level joins. **AsterixDB has a lower bound of 10-20ms for any query due to its operation mode (not optimized for simple or non-cluster queries). ^{OOM}Out of memory error in the application-level code. ^{TO}Timed out.

Figure 4: Query runtime (seconds) on large dataset (single node, 15 minutes timeout).

PostgreSQL and CrateDB. For Q9 the application level sort-based GROUP BY operator could not store the number of rows fetched in memory what resulted in an out of memory error. GridDB's performance drops significantly with mapping T since it is not able to use the primary index on any of the queries, while InfluxDB performs slightly better than GridDB due to its faster read performance (see Q3-Q5).

CrateDB, a column oriented DBMS (with the benefit of having indexes on all the columns involved in the selection predicate and not just timestamp column), performed best on Q7 and Q9. It took a considerable amount of time on Q8, as it failed to come up with an optimized query plan (it tried to do selection after join) and did not perform well on Q10 as the query involved most of the table columns.

All the queries on MongoDB perform better with mapping A2 which suggests that a join (lookup) with smaller metadata tables is many times better than having bigger nested documents in our use case (scanning data based on a time range). Queries that involve joins of two big collections (e.g., queries Q7 and Q8) timed out since the join (lookup) operator in MongoDB is limited in functionality and it failed to push selections down in its aggregation pipeline. AsterixDB supports full SQL functionality, has a more advanced query optimizer, and better JOIN support compared to MongoDB which made its performance much better. PostgreSQL outperformed AsterixDB as the latter has to scan rows which are comparatively larger in size and the former came up with a better query plan since it has a more mature optimizer and it stores statistics about the data. SparkSQL, even with columnar storage (parquet files on HDFS), did not perform well since it has to scan the entire dataset for all the queries due to the lack of support for secondary indexes.

Window based query (Q11). For Q11 we used the best mapping configuration per system according to the previous results. PostgreSQL performed best because of its superior query optimizer and execution engine. Even when GridDB, InfluxDB, and MongoDB had the added overhead of the window task performed outside of the database, they performed better than SparkSQL since it does not support secondary indexes. Also, since the difference in query execution time for these database systems only depend upon the time to fetch the filtered data from the DB (rest of the computation is done in the application side), the performance trend for these systems follow the trend described for Q3-Q6.

Experiment 3 (Application vs. Database Joins - Figure 3): Experiment 2 showed that GridDB and InfluxDB, outperformed systems with native join support using application level joins. We compared them further by varying query selectivity levels. We selected PostgreSQL and Q8 to compare native vs. application-level join as comparing across different systems would make it difficult to determine whether the performance is due to the type of join or other factors. To implement application-level joins, we first send a selection query to the outer presence table, and then, for each row in the result set, a selection to inner presence table.

As expected, native joins outperformed application-level joins (see Figure 3) due to the higher overhead of the latter (e.g., multiple independent queries compiled separately). For the native join case, PostgreSQL selected, for low selectivities, a plan consisting of an index scan on the timestamp predicate of the outer presence table followed by a nested loop index join with the inner presence table and, for higher selectivities, a sequential scan of the outer presence table followed by a nested loop index join. For the application-level join case, it selected the index scan for the outer table at low selectivities, changing to a sequential scan at higher selectivities. All queries on the inner presence table used an index scan on the join column value for all selectivities.

The results show that for very small selectivities (below 0.5 percent of the dataset) application-level joins perform very competitively to native joins. This is interesting since in IoT applications joins are often between small metadata tables and large timeseries data and queries can be very selective, filtering data corresponding to a small time range. In such contexts, simpler timeseries databases such as InfluxDB and GridDB (that typically do not support joins) could outperform more complex systems by relying on external application-level joins.

Experiment 4 (Effect of Time Ranges - Figure 6): Time is a fundamental component of IoT data and queries. We explore further the effect of varying time ranges in queries w.r.t. systems' performance. We chose Q6 and Q8 and varied their associated time ranges to: one day, one week, two weeks, one month, and two months. GridDB, InfluxDB, and CrateDB continue to outperform other systems on Q6 (see Figure 6 on the left), although their runtimes increase with the selectivity. PostgreSQL performed as well as the timeseries database systems for low selectivities on Q6, but its performance suffers due to increased secondary index lookups as selectivity increases. SparkSQL performance was not affected since it always performs a table scan. The relative performance of the document stores on Q6 gets worse with increased selectivity due to their comparatively larger record sizes. On Q8 (see Figure 6 on the right), AsterixDB, SparkSQL, and CrateDB chose a hash join based approach and took almost the same time for all selectivity values. PostgreSQL, on the other hand, with its mature query optimizer and statistics, chose an index nested loop join, which

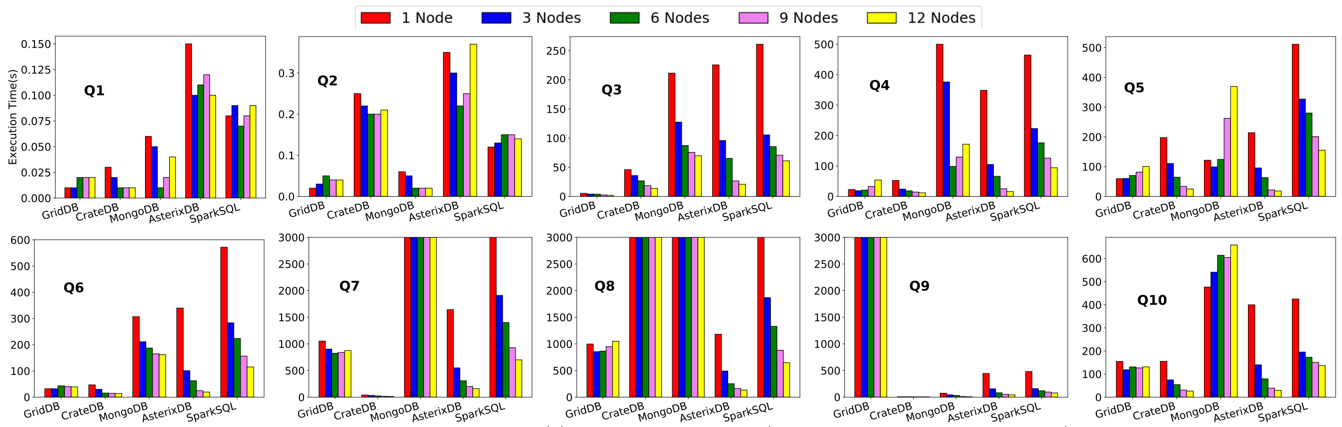


Figure 5: Query runtime (s) on small dataset (multi node, 1 hour timeout).

performed well on low selectivities but dropped with with increase in selectivity. Timeseries databases, using our index nested loop based application-level joins, performed quite well for low selectivity values, but their performance grew super-linearly with increasing selectivity.

Experiment 5 (HDD vs. SSD - Table 5): The bulk of our experiments was performed in a cluster with hard disks. We performed an additional experiment to explore the impact of SSDs. To this end, we setup two AWS EC2 instances with same specifications: one with a general purpose SSD and another one with throughput optimized HDD. SSD's provide performance improvement over HDDs on sequential reads and writes, however, they are mainly optimized for random reads and writes and provide order of magnitudes faster IO per second compared to HDDs [33]. We performed insert and query performance test for the single node large dataset (for the best performing mapping in our previous experiments). For all the DBMSs (see Table 5) the insert throughput increased because of the superior write performance of SSDs. However, as there were very few random updates and most of the IO performed was sequential and not random, the increase in throughput was limited. For Q6 every system experienced a decrease of execution time due to SSD's higher IOPS. However, the improvement for InfluxDB and GridDB was limited since the data is arranged based on timestamp which implies fewer random updates. For query Q10, all the systems showed a decrease of about 2 times in execution time with SSD since Q10 involves scanning the dataset and therefore the reads are mostly sequential.

CPU and IO Usage: We analyzed the percentage of execution time than went into CPU and IO tasks per query (plots are included in the extended version of the paper [10]). In cases where the query timed out or threw an out of memory error, we analyzed the CPU/IO breakdown before that. For *Metadata Queries* Q1 and Q2, both the CPU and IO utilization is very low and balanced across DBMS, since these queries run in few milliseconds. For *Simple Selection and Roll Up* queries Q3-Q6, almost all the systems spent more time on IO operations. CrateDB and SparkSQL spent more time in CPU as they need to decompress and de-serialize after reading data from disk. For *Complex queries* Q7-Q10, AsterixDB spent a higher amount of time on IO than other systems, as it requires scanning comparatively larger records. InfluxDB and GridDB have smaller record size and support compression, hence they spend comparatively less time in IO and have CPU as their bottleneck. Additionally, part of the higher CPU utilization is due to the implementation of unsupported Join and Aggregation operations in the application domain. SparkSQL spends some time doing IO (since it does not support secondary indexes and scans the complete table) but since it needs to perform de-serialization of data [39], it spends longer time on CPU than other systems, making CPU the bottleneck. CrateDB spends considerably less time doing IO as it uses indexes and columnar compression, causing CPU to be the bottleneck because of the added cost of decompression. PostgreSQL also, had CPU as the bottleneck for queries Q7-Q10.

Table 5: Ingestion throughput and query latency.

DBMS	Ingestion (row/s)		Q6 (s)		Q10 (s)	
	HDD	SSD	HDD	SSD	HDD	SSD
griddb	40,610	54,973	0.72	0.43	**2.40**	**1.02**
postgresql	18,300	28,391	3.71	0.75	4.12	2.45
mongodb	12,543	15,408	5.17	1.55	47.80	24.90
asterixdb	17,100	26,907	7.92	3.26	8.24	4.58
cratedb	4,620	5,400	0.25	0.15	6.59	3.04
influxdb	**66,670**	**71,428**	**0.12**	**0.08**	4.03	2.60
sparksql	-	-	12.93	5.41	10.40	4.85

5.2 Multi-Node Experiments

For multi-node experiments we used larger datasets (see Table 3). Data was partitioned over 1, 3, 6, 9, and 12 nodes (each node is an Intel i5 CPU, 8GB RAM, 800GB HDD, and CentOS 7 machine connected via a 1 Gbps network link). The DBMS instances on each node have the same configuration as in the case of the single node setup. We skipped PostgreSQL and InfluxDB for multinode experiments since the former does not support horizontal sharding natively and the latter supports sharding features only in its enterprise edition which is not open source. For the remaining systems, we chose their most performant mapping for our workload, inferred from the results of the single node experiments. MongoDB, CrateDB, allow data to be partitioned on any arbitrary key, so we partitioned the observation data based on the sensor-id and the semantic observation data on the semantic entity-id. For AsterixDB, data is partitioned on the primary key, as it uses hash-partitioning on the primary key for all datasets. In GridDB, a container is stored fully on a single node since GridDB does not provide an explicit partitioning method. GridDB balances data across nodes by distributing different containers to different nodes based on the hash of their key/name. The information regarding the allocation of containers to nodes is populated to all the nodes in the cluster, making it easy for the client library to locate any container.

Experiment 6 (Insert Performance - Table 4): Similar to Experiment 1, GridDB performs well on inserts as expected, although the per tuple insertion time increased w.r.t. the single node case even with multiple nodes writing data in parallel; the data size per node has increased by 2.5 times, causing more data flushes from memory to disk compared to the single node case. MongoDB's per tuple insertion time also increased with respect to the single node experiments. Since each tuple is now required to be routed by *mongos* service to the appropriate node based on

Table 6: Query runtimes (s) on large dataset (12 nodes).

Query	cratedb	mongodb	asterixdb	sparksql
Q1	**0.02**	**0.02**	0.08	2.5
Q2	1.14	0.7	**0.67**	3.8
Q3	**42.83**	239.95	95.16	264.86
Q4	**44.28**	285.17	97.63	255.46
Q5	**89.25**	494.50	98.08	305.40
Q6	**35.72**	310.85	90.35	292.76
Q7	**60.93**	NA	924.66	1, 245.57
Q8	TO	NA	**876.44**	1, 197.45
Q9	**12.18**	30.46	162.13	180.77
Q10	103.23	2, 235.8	**80.73**	251.35

the sharding key, it was not able to make use of the batched insert, causing its insertion time to increase. AsterixDB's per tuple insert time also increased, but, with this larger dataset, we started seeing the benefits of write optimized LSM-trees as its write performance got considerably better than MongoDB. CrateDB, because of its columnar storage, took the most amount of time in the insert tests.

Experiment 7 (Query Performance - Figure 5 & Table 6): Figure 5 shows the query performance results while Table 6 shows the results for a 12 node configuration for the large database with 1.6 billion rows[4]. Variance for each query template for multi-node setup is available in the extended version of the paper [10].

Since every query in GridDB can only involve one container, its query processing happens only on a single node. GridDB remains better compared to other databases for queries Q3-Q6 for upto 3 node setup. Its performance did not improve with an increasing number of nodes except for query Q3 which requires data to be fetched from only one node[5]. For other queries, that require data to be fetched from multiple containers, GridDB's performance does not improve since it natively executes only single container queries and, thus, query processing happens only at a single node. Since the application code we wrote initially to execute queries in GridDB was a sequential program, GridDB was not able to leverage any parallelism from the multi-node setup. We implemented multi-threaded programs to fetch data from multiple containers simultaneously for GridDB. The query times for the parallel version are available in the long version at [10]. This improved the query performance for GridDB considerably, especially for queries Q3-Q5, as these queries just fetch data from multiple containers based on conditions, without any reduction step due to aggregation type. We did not see much improvement on queries Q7 and Q8.

For the CrateDB cluster, the sharding information is available at all the nodes each of which runs a query execution engine that can handle a distributed query (involving distributed joins, aggregations etc.). However, only the metadata primary node can update this information. An application can send queries to any of the nodes in the cluster. CrateDB again performed well on complex queries (Q7, Q9, Q10) due to its columnar storage. Also, CrateDB was able to scale well with its performance improving with increasing number of nodes for all queries.

An AsterixDB cluster consists of a single controller node and several worker nodes (storing partitioned data) called node controllers. Applications send queries to the cluster controller, which converts the query into a set of job descriptions. These job descriptions are passed to the node controllers running a parallel data flow execution engine called Hyracks [20]. The Cluster controller reads the sharding information and other metadata from a special node

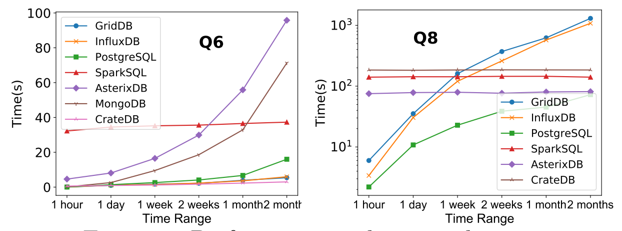


Figure 6: Performance with time selectivity.

controller called the metadata node controller. Among all the databases, AsterixDB scaled the best, with its performance improving significantly with increasing numbers of nodes. AsterixDB outperformed every other database on queries Q5 and Q8 for the 9 node cluster configuration. For the smallest queries AsterixDB suffered from the overhead of its job distribution approach.

In a clustered setting, MongoDB uses a router process/node called mongos that fetches information about the shards from a centralized server (possibly replicated) called the config server. The application sends its query to the mongos process, which processes the query and asks for data from respective shards (in parallel if possible) and does the appropriate merging of data. Even with the mongos service, MongoDB does not support joins between two sharded collections, so we skipped queries Q7 and Q8. MongoDB stores all unsharded collections together on the same shard called the primary shard. MongoDB was not able to scale as well as AsterixDB, with many queries not able to benefit from multiple nodes being able to work in parallel. Furthermore, for queries Q5 and Q10, its performance actually degraded with an increasing number of nodes, as its optimizer started to pick a collection scan over an index scan.

CPU and IO Usage CPU and IO usage for multi node setups with detailed explanations are available in the extended version of the paper [10]. The results (representing the percentage of the total query time spent in CPU and IO on all nodes –master and 3 workers–) are similar in nature to the results for the single node setup. Queries Q1-Q6 are IO bound and Q7-Q10 are CPU bound on most DBMSs.

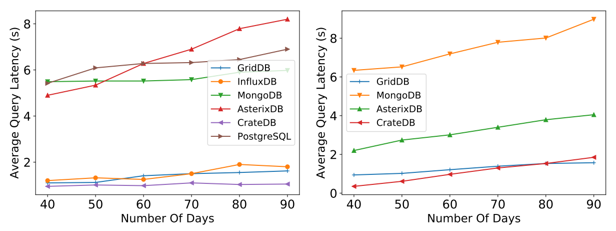
5.3 Mixed Workloads Experiments

We compare system performance under the online mixed workload where queries of the same template are executed in parallel with data ingestion. We used two different levels of data insertion rate, slow and fast, where data is available to insert at the rate of 10,000 and 50,000 observations per second, respectively. The experiment was repeated 6 times with different sizes of data (based on days) already ingested. The experiment starts from a database state where data of a varying number of days is already ingested. The inserts and queries are then done in parallel (multiple threads). In order to make the queries consistent (return the same result) across different database systems (even if they support a lower ingestion throughput than required), we generated query instances that have a time range corresponding to the dataset ingested prior to the time of the insert commands - that is, the queries retrieve data that had already been inserted at the beginning of the experiments.

Experiment 8 (Online inserts and queries - Figure 7): Figure 7 shows the average query latency for Q6 w.r.t. the number of days for slow data generation rate on a single node as well as a multi node (3 node) setup. The query latency is increasing with increasing number of days for all the database systems, as the data size is increasing. The latency increase rate is comparatively higher in case of faster insert rate as expected since the queries are now running in parallel with a much higher load of inserts. The results for

[4]We do not include results for GridDB on the 12 node configuration since inserting the large database would take over 50 hours (due to a lack of bulk insertion and insertion rate of 10k tuples/second).

[5]GridDB performance for Q3 improves since data per node reduces as the number of nodes increase.



(a) Single Node (b) Multi Node

Figure 7: Q6 in mixed workload (slow insertion rate).

multi-node version of the experiment show a similar scalability trend for the DBMSs as discussed in Experiment 7, with AsterixDB, CrateDB showing lower query latency with multi node setup, whereas GridDB and MongoDB did not show much improvement. The same relative performance of the systems is observed for the fast data generation rate (the plot is included in the extended version of the paper [10]).

Experiment 9 (Continuous Query - Figure 8): We perform an experiment with mixed workload and the sliding window continuous query (CQ). We set the window length to 10 seconds and the length of the sliding to 5 seconds. Since none of the DBMSs support stream processing, we implemented the continuous query logic on the application side. We buffered occupancy data of all the rooms in the last 10 seconds time window, followed by running a selection query on the DBMS to discover the rooms of type "Lecture Hall" and finally joined it with the buffered data. Figure 8 shows the results for slow and fast data generation rate on a single node as well as a multi-node (3 node) setup. Since the query is executed as part of the application, the difference in performance can be attributed to the DBMS ingestion rate supported and performance of a simple selection query on a static table. Hence, GridDB and InfluxDB performed significantly better because of their better ingestion performance (discussed in more detail in Experiment 1).

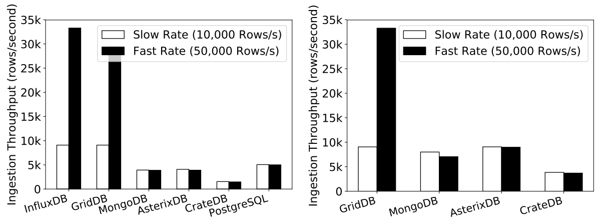
6. CONCLUSION

The design of SmartBench and the analysis of the performance results have lead to several interesting observations to us: 1) In an IoT system, the data exhibits a lot of temporal and spatial correlations among different entities and events. Application queries are posed on such correlations as well as on time-varying sensor data. 2) The mapping of heterogeneous sensor data to the database representation plays a critical role for ingestion and query performances. 3) The complexity of IoT query workload varies widely. It ranges from simple selections on temporal attributes to multiway-joins, grouping, and aggregations. Depending on application context, efficient application level joins can be devised. 4) An IoT system must support a high rate of data arrival and thus databases with higher ingestion rates are more applicable. Data follows an append-only pattern with rare updates. The volume of data can be very large and hence scale up and scale out functionalities of databases are required.

We highlight some key observations about the suitability of DBMSs and implementation choices for IoT workloads:

- Specialized timeseries databases are suited for sensor data (fast insertion and time-based selection queries) but they do not provide natural ways to store other data needed to build IoT applications (e.g., spatial relationships, entities, events). For instance, InfluxDB does not provide any way to store non-timeseries data (e.g., metadata). One can overcome such limitations by storing such information in a different database and appropriately co-executing a query across both systems (as we did for InfluxDB using PostgreSQL).
- Document stores are suited to represent heterogeneous data but an embedded representation comes at a high cost in terms of performance compared to a normalized representation. For instance, queries with foreign key based joins, on datasets with normalized documents, run faster compared to queries without foreign key based joins with large denormalized documents. This holds even in a multi-node setting where the smaller collection may not be even present on the same node. Thus, a system that supports document level specification, but (semi)-automatically maps such data to an underlying structured representation could offer the best of both worlds. Examples of such a strategy are closed datasets in AsterixDB and JSON shredding in Teradata [11].
- Time series databases, such as InfluxDB and GridDB, performed well on inserts, simple selection queries, and several complex join queries by exploiting application-level joins. This suggests an opportunity to write wrappers that split SQL queries into a set of queries that can be executed directly on such a system, and continue the remainder of the query execution using application-level operators (e.g., application-level joins). Such a wrapper could provide a timeseries database with the capability of executing full SQL and still being better in terms of performance in situations where at least one of the tables being joined is small, perhaps, due to selection, as is the case in SmartBench.
- Traditional relational database systems like PostgreSQL do well on both insert and query performance on single node but do not scale horizontally. Document stores, while they scale easily (specifically AsterixDB, which performs very well with a large cluster), have query performance that is not as good as a mature relational system on a single node.
- UDF technologies supported by today's databases are not adequate to enrich data during ingestion. Enrichment, today, is performed outside the database (e.g., in application code, or through a streaming engine) during ingestion. Such an architecture can be sub-optimal specially if complex enrichment function need to run queries to retrieve past data [48]. Co-optimizing enrichment with ingestion (e.g., through batching, or selectively choosing which enrichment to perform in real-time, and which to do progressively, etc.) is an important challenge to support real-time smart applications.

Finally, our key observation (based on the discussion above) is that, like in other domains, while different systems offer different advantages, there is no single system that offers the "best" choice. The emerging field of Polystores [26], which aims to provide integration middleware allowing applications to store different parts of their data in different underlying databases, may be a relevant solution.



(a) Single Node (b) Multi Node

Figure 8: Insertion throughput (CQ).

Acknowledgements

This material is based on research sponsored by DARPA under agreement number FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work was also supported by NSF award CNS-1925610 and Toshiba. We also want to thank Fangfang Fu for her help in the development of the data generator tool.

7. REFERENCES

[1] Apache Kafka, Stream Processing Engine. https://kafka.apache.org/intro. [Online; accessed June-2020].

[2] Apache Storm, Stream Processing Engine. http://storm.apache.org/about/integrates.html. [Online; accessed June-2020].

[3] Couchbase NoSQL Database. https://www.couchbase.com/. [Online; accessed June-2020].

[4] DB-Engines Ranking. https://db-engines.com/en/ranking. [Online; accessed June-2020].

[5] GridDB, NoSQL Database System For IoT. https://griddb.net/en/. [Online; accessed June-2020].

[6] IBM DB2 Event Store. https://www.ibm.com/products/db2-event-store. [Online; accessed June-2020].

[7] InfluxDB, Timeseries Database System. https://www.influxdata.com/_resources/. [Online; accessed June-2020].

[8] MongoDB. https://www.mongodb.com/. [Online; accessed June-2020].

[9] PostgreSQL, Relational Data Management System. https://www.postgresql.org/. [Online; accessed June-2020].

[10] SmartBench: A Benchmark For Data Management In Smart Spaces. Technical Report, UCI, 2019. http://github.com/ucisharadlab/benchmark.

[11] Teradata. http://www.teradata.com. [Online; accessed June-2020].

[12] TIPPERS. http://tippersweb.ics.uci.edu/. [Online; accessed June-2020].

[13] TPC-C Bechmark. http://www.tpc.org/tpcc/. [Online; accessed June-2020].

[14] The Internet of Things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010.

[15] The SSN ontology of the W3C semantic sensor network incubator group. *Journal of Web Semantics*, 17:25 – 32, 2012.

[16] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A scalable, open source bdms. *PVLDB*, 7(14):1905–1916, 2014.

[17] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *13th Int. Conf. on Very large data bases-Volume 30*, pages 480–491. VLDB Endowment, 2004.

[18] M. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver. Iotabench : an internet of things analytics benchmark. In *6th ACM/SPEC Int. Conf. on Performance Engineering*, pages 133–144. ACM, 2015.

[19] 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 *2015 ACM SIGMOD Int. Conf. on Management of Data*, pages 1383–1394. ACM, 2015.

[20] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *2011 IEEE 27th Int. Conf. on Data Engineering*, pages 1151–1162. IEEE, 2011.

[21] M. Botts, G. Percivall, C. Reed, and J. Davidson. OGC sensor web enablement: Overview and high level architecture. In *Int. Conf. on GeoSensor Networks*, pages 175–190. Springer, 2006.

[22] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[24] T. P. P. Council. Tpc-h benchmark specification. *Published at http://www. tcp. org/hspec. html*, 21:592–603, 2008.

[25] U. Dayal, C. Gupta, R. Vennelakanti, M. R. Vieira, and S. Wang. An approach to benchmarking industrial big data applications. In *Workshop on Big Data Benchmarks*, pages 45–60. Springer, 2014.

[26] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The bigdawg polystore system. *ACM Sigmod Record*, 44(2):11–16, 2015.

[27] 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 *2013 ACM SIGMOD int. Conf. on Management of data*, pages 1197–1208. ACM, 2013.

[28] C. Gormley and Z. Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine.* " O'Reilly Media, Inc.", 2015.

[29] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Acm Sigmod Record*, volume 23, pages 243–252. ACM, 1994.

[30] L. Gu, M. Zhou, Z. Zhang, M.-C. Shan, A. Zhou, and M. Winslett. Chronos: An elastic parallel framework for stream benchmark generation and simulation. In *2015 IEEE 31st Int. Conf. on Data Engineering (ICDE)*, pages 101–112. IEEE, 2015.

[31] J. E. Hoag and C. W. Thompson. A parallel general-purpose synthetic data generator1. In *Data Engineering*, pages 103–117. Springer, 2009.

[32] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th Int. Conf. on Data Engineering Workshops (ICDEW)*, pages 41–51. IEEE, 2010.

[33] V. Kasavajhala. Solid state drive vs. hard disk drive price and performance study. *Proc. Dell Tech. White Paper*, pages 8–9, 2011.

[34] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *2014 IEEE/ACM 7th Int.*

Conf. on Utility and Cloud Computing (UCC), pages 69–78. IEEE, 2014.

[35] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth. Machine learning for internet of things data analysis: a survey. *Digital Communications and Networks*, 4(3):161 – 175, 2018.

[36] D. Massaguer, S. Mehrotra, R. Vaisenberg, and N. Venkatasubramanian. Satware: a semantic approach for building sentient spaces. In *Distributed Video Sensor Networks*, pages 389–402. Springer, 2011.

[37] R. O. Nambiar and M. Poess. The making of tpc-ds. In *32nd Int. Conf. on Very large data bases*, pages 1049–1058. VLDB Endowment, 2006.

[38] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ query language: Configurable, unifying and semi-structured. *arXiv preprint arXiv:1405.3631*, 2014.

[39] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 293–307, 2015.

[40] P. Pirzadeh, M. J. Carey, and T. Westmann. Bigfun: A performance study of big data management system functionality. In *2015 IEEE Int. Conf. on Big Data (Big Data)*, pages 507–514. IEEE, 2015.

[41] M. Poess, R. Nambiar, K. Kulkarni, C. Narasimhadevara, T. Rabl, and H.-A. Jacobsen. Analysis of TPCx-IoT: The first industry standard benchmark for iot gateway systems. In *2018 IEEE 34th Int. Conf. on Data Engineering (ICDE)*, pages 1519–1530. IEEE, 2018.

[42] A. Shukla, S. Chaturvedi, and Y. Simmhan. Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257, 2017.

[43] J. M. Stephens and M. Poess. Mudd: a multi-dimensional data generator. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 104–109. ACM, 2004.

[44] Y. Tay, B. T. Dai, D. T. Wang, E. Y. Sun, Y. Lin, and Y. Lin. Upsizer: Synthetically scaling an empirical relational database. *Information Systems*, 38(8):1168–1183, 2013.

[45] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[46] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. Nexmark–a benchmark for queries over data streams (draft). Technical report, Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.

[47] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th Int. Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499. IEEE, 2014.

[48] X. Wang and M. Carey. An idea: An ingestion framework for data enrichment in asterixdb. *PVLDB*, 12(11):1485–1498, 2019.

[49] M. E. Yazid Boudaren, M. R. Senouci, M. A. Senouci, and A. Mellouk. New trends in sensor coverage modeling and related techniques: A brief synthesis. In *2014 Int. Conf. on Smart Communications in Network Technologies (SaCoNeT)*, pages 1–6, 2014.