# SmartBench: A Benchmark For Data Management In Smart Spaces

Peeyush Gupta, Michael J. Carey, Sharad Mehrotra, Roberto Yus

University of California, Irvine

## ABSTRACT

With advances in the Internet of Things (IoT), several database technologies (e.g., time series DBs, document stores, cluster computing frameworks, etc.) have begun to advertise themselves as systems specially suited for managing IoT data. The few IoT database benchmarks that have been developed to compare systems primarily focus on support for fast sensor data ingestion. Such benchmarks ignore the complex queries that are beginning to be a part of real-world IoT deployments, especially those that deal with semantically meaningful interpretation of raw sensory data. We propose SmartBench, a benchmark focusing on complex analytics queries over 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 diverse and representative database systems and highlight some interesting findings that can be considered when deciding what database technologies to use under different types of query workloads.

## 1. INTRODUCTION

The emerging IoT revolution [12] 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, in turn, can be used to implement new functionalities and/or bring transformative improvements to existing systems. Central to the design and implementation of IoT applications is the data management technology that can represent, store, and query sensor data. Given the importance of IoT, a multitude of database systems, whether they be standard relational systems, key-value stores, document databases, or specialized systems such as time series stores, have begun branding themselves as being suitable for IoT applications.

IoT systems impose special requirements on database technology. IoT data can be voluminous as well as be generated at very high speeds – even a simple smart phone contains dozens of sensors that can generate data continuously every few seconds, a medium office building has several thousand HVAC sensors producing data at a similar rate, and a university campus or an office campus may consist of several hundred thousand such sensors. Sensor data is typically time-varying arrays of values and, often, queries over sensor data involve temporal operators and aggregations. Typically, in an IoT setting, the underlying system has to deal with heterogeneous types of sensors and their observations as new sensors/devices are being introduced into the market with very diverse capabilities (e.g., from cameras, thermal sensors, GSR sensors, wearable technologies, smart fabrics, Arduino platforms, flora, etc.). Finally, the sensor data captured is generally too low-level for final applications, and the underlying database system must support mechanisms to translate such data 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.

While different database technologies address different requirements of IoT applications, none seems to provide a complete solution to all the challenges. Big Data management technologies built on top of cluster computing frameworks, like Hive [37] and SparkSQL [16], provide efficient ways to run complex data warehouse (OLAP) types of 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. contrast, stream processing systems like Apache Kafka [1], Apache Storm [2] and Apache Flink [**?**] 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 [8], can make use of indices and better join and aggregation operators, but they do not scale well to huge amounts of data and inherently do not support storing heterogeneous data. Document stores like MongoDB [7], Couchbase [3], and AsterixDB [13] 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 [6] and GridDB [5], provide fast ingestion rates and fast selection on time ranges but they fail to perform complex queries due to limited functionality.

The database community has traditionally relied on benchmarks to understand the trade-offs between different database systems (e.g., the TPC-H [19] benchmark has been widely adopted by both industry and academia). Due to the specific requirements of IoT data management mentioned above, traditional benchmarks are not suitable for this scenario. Because the IoT area is still in its early stages, there are only a few IoT database benchmarks in the literature (e.g., [33, 20, 15, 34]), the focus of which have been mainly on testing the performance of different database systems for fast ingestion of sensor data. Although the ingestion rate is an important challenge, the suitability of a database system for the IoT domain depends upon other aspects as well. In particular, existing benchmarks have overlooked the issue of evaluating databases for their support for complex analytical queries that can arise in practical IoT settings.

We present a benchmark that focuses on the evaluation of database systems' support for data processing through complex queries. SmartBench is derived from a real-world smart building monitoring system currently deployed in several smart buildings of the campus of the University of California, Irvine [10]. The benchmark is comprised of an IoT data model, a set of representative queries, and a tool to generate synthetic IoT datasets. The IoT benchmark 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 which can be mapped to the data model supported by the underlying database system. We present ten representatives queries for IoT domains that deal with the fetching of sensor data to support analytical tasks, as well as the transformation of such raw data into semantically meaningful information. Finally, we present 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.

In summary, the main aspects of our benchmark are: 1) It is motivated by the requirements of IoT applications (deployed campus-wide) on a data management system; 2) It compares the performance of database systems for queries on raw sensor data as well as semantic data and its relationship with the sensor data, giving a holistic view of the efficiency and capabilities of different database systems in a real life IoT system; 3) Although it is based on a smart building schema, 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; 4) It not only compares different database systems with different technologies, but also compares different ways to map data into each database system; 5) It also tests the scalability (both scale up and scale out) of different database systems by comparing their performance under different data loads for a single node as well as a multi-node setup.

Finally, 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 databases, document stores, and cluster computing frameworks). 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 database technology(ies) for their data management needs and guide future database developers to support such needs.

## 2. RELATED WORK

Several benchmarks have been developed to test database systems from diverse perspectives. The most widely used benchmarks are TPC-H [19] and TPC-DS [30], which were designed to test a database system's analytical data processing and data warehousing (OLAP) performance under different data workloads. TPC-C [11], is the industry standard for online transaction processing (OLTP). These benchmarks, however, all focus on comparing relational database systems in the context of traditional data management scenarios. As explained before, IoT environments introduce a set of challenges that require different technologies and therefore different benchmarking strategies.

**IoT benchmarks.** Recently, several benchmarks have been developed in the IoT context. The TPCx-IoT [33] 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. It comes as a toolkit that generates a workload consisting of concurrent inserts and simple range queries. It is more concerned with the end-to-end performance of IoT gateway systems, irrespective of the database systems these gateways may use. IoTABench [15] is a benchmark for a smart meter use case and includes queries that focus on mainly on computations in such a scenario (e.g., bill generation). RIotBench [34] is a benchmark based on real world IoT applications that aims at evaluating systems using streaming time-series data workload. Neither RIoTBench nor IoTABench consider the OLAP aspect of an IoT system and are instead more focused on data ingestion, streaming data, and continuous queries. Finally, Dayal et al. [20] 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) [14] and StreamBench (distributed streaming) [28], are focused on testing the performance of real-time data insertion/processing and queries. In contrast, big data benchmarks like BigBench [22], BigFUN [32], HiBench [27], and BigDataBench [38] compare systems on their ability to process a large volume of heterogeneous data. BigBench [22] provides a semi-structured data model along with a synthetic data generator that mimics the variety, velocity and volume aspects of big data systems. Also, BigBench queries cover different categories of big data analytics from both business and technical perspectives. BigFUN [32] is a benchmark based on a synthetic social network scenario with a semi-structured data model. HiBench [27] and BigDataBench [38] 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. 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.

**Data generation tools.** Benchmarks typically offer tools to generate datasets of varying size to test systems in different situations. Most of these synthetic data-generating tools are application specific and, hence, not reusable for our purposes. For example, IoTABench [15] provides a Markov chain model based synthetic time-series dataset generator specific to smart metering scenarios that cannot be generalized to other IoT scenarios. Some data generators provide mechanisms to generate datasets for other applications (e.g., [24, 35, 26, 36, 25]). Gray et al. [24] developed ways to generate large amounts of data in parallel with different user-provided distributions. MUDD [35] and PSDG [26] scan a complete user-provided dataset to compute distributions and dependencies in the data and then generate synthetic data with the computed distributions. UpsizeR [36] uses clustering algorithms to find foreign key based correlations but assumes that non-key attributes depend only upon the key attributes. Chronos [25] can generate synthetic streaming time series data, preserving temporal dependencies and column-wise correlations. However, the tools mentioned above cannot be used to generate generic IoT data that maintains the temporal and spatial correlations of real-world data and thus, they cannot scale up IoT data in a semantically meaningful way. Our data generator tool can generate data for heterogeneous sensors preserving the temporal and spatial correlations of the data generated.

# 3. SMARTBENCH BENCHMARK

Before we describe the SmartBench[1], we first highlight some of the design goals that guided our design.

## 3.1 Design Criteria for SmartBench

Sensors are becoming increasingly pervasive, with a single smart environment housing a large variety of sensors ranging from video cameras, microphones, thermostats, smart TVs/assistants/refrigerators, and even WiFi Access Points (which can sense which devices are connected to them). Sensors 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:* Database 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 database system 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 need to be processed/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*: Database 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*: Database technology needs 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 database 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.

## 3.2 Schema

The schema used in SmartBench is based on the Observable Entity Relationship (OER) Model introduced in [29] which, in turn, is based on SensorML [18]. OER is an extensible model that allows incorporating new/heterogenous types of sensors, observations, spaces, and users. The complete data model that we use in the benchmark is specified in JSON format in Appendix B. In the discussion below, we highlight key entities and concepts in a smart space. To structure the discussion, we categorize entities along into three interrelated layers: devices, observations, and domain entities.

**Device Layer.** Devices (aka, sensors and actuators) can, in general, be either physical or virtual. Given that our goal is to design an IoT analytic benchmark, we focus the discussion on sensors. *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) to produce presence infor-

[1]The SmartBench code, data generation tool, and a longer version of this paper can be accessed at `http://github.com/ucisharadlab/benchmark`.

mation. 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* and *coverage*, where sensor type dictates the type of observation the sensor generates (e.g., an occupancy detector generates a count of people). *Coverage* corresponds to the spatial region in which the sensor can capture observations. For physical sensors, coverage is simply a function of its location (which can change with time for mobile sensors) - e.g., the coverage of a camera is its view frustum. For virtual sensors, coverage is a function of the coverage of the sensors that generated data which are input to the virtual sensor - 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 and interact with the space and with each other. 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*. 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) changes with time. Dynamic attributes are *observable* if they can be associated with one or more (physical or virtual) sensors whose observations can be used to determine the value of the attribute. 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 be used to determine the occupancy of Room 2065 at time t1. Likewise, *Inv_Coverage(John, t2, location)* will provide a list of (virtual) sensors that can provide John's location at time t2. The inverse coverage for a given type of observation (e.g., occupancy, location, temperature, etc.) is computed by running an appropriate query on the coverage information about sensors.

To implement the above model in existing database systems, 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 4.1.

## 3.3 Queries

The benchmark consists of ten representative queries motivated by the the need to support diverse applications as mentioned in Section 3.1. 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 other 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. 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.

- **(I) Insert**: Increases the sensor observation (either raw or semantic) data by a percentage (e.g., in our experiments by 10%). Every IoT system needs to continuously ingest streaming sensor observations, hence, insert performance is very important for such systems.
- **(Q1) Coverage(**$s \in Sensors$**)**: returns the coverage of a given sensor $s$. Such queries are called every time lower-level raw sensor data is transformed/enriched into semantic observations, as is often done as part of a pre-processing/caching step in order to support the efficient evaluation of queries over semantic data.
- **(Q2) InverseCoverage(**$L$**,** $\tau$**)**, where $L$ is a list of locations, and $\tau$ is a sensor type: lists all sensors that can generate observations of a given type $\tau$ 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 instatiations 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(**$\tau$**,** *cond*, $t_b$**,** $t_e$**)**: is a conditional

observation query that selects observations generated by sensors of given type $\tau$ 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 $\theta$ 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**($u \in Users$, $\eta$, $t_b$, $t_e$): retrieves the average time spent per day by subject $u$ in locations of type $\eta$, (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**($L$, $\Delta_t$, $t_b$, $t_e$): retrieves the occupancy level for each location in the list $L$ every $\Delta_t$ 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.

The set of queries listed above represent 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.

## 3.4 Data And Query Generator

To enable the benchmark to test database system performance for IoT applications at different scales, we created a synthetic data & query generation tool (see Figure 1). The generation tool takes real data and metadata from an instrumented smart building collected over a period as a seed to generate a user specified amount of synthetic IoT data. The tool can be used to both scale up or down the smart building data. Similar to relational data generator tools (e.g., UpsizeR [36]) our data generation tool preserves the spatial and temporal correlations in the seed data to support realistic selectivities for both selection and join queries.

**Metadata Generation.** The seed data, in addition to containing real sensor data, also contains metadata about the building, description of the space, the different types of sensors, their locations, the set of users, etc. The dataset

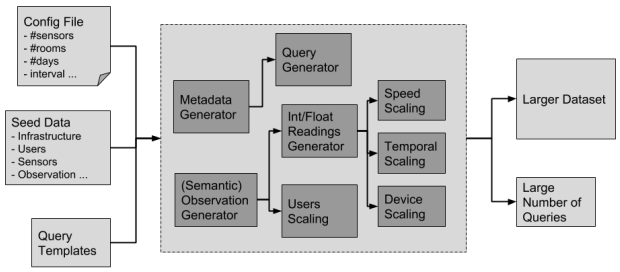



Figure 1: Data and query generator schematic diagram.

generator uses parameters specified in a configuration file (e.g., number of users, number of sensors of each type, etc.) to scale up the metadata[2]. Synthetic user metadata, such as the attributes of the user like id, name, and email id, are generated randomly. In particular, the group name attribute is chosen uniformly at random from a set of available user groups. Sensors are generated keeping the spatial constraints in mind. For instance, the location attribute of a sensor is chosen uniformly at random from the list of available locations provided in the seed data file.

**Sensor Data Generation.** Sensor data is scaled up in one of the following ways: 1) Increasing the duration of data capture; 2) Increasing the data generation frequency of sensors; or 3) Adding synthetic data generated by synthetic sensors. These three methods can be combined in different orders to create a dataset with a required number of sensors, duration, and frequency. Our data generation tool allows all these parameters to be configured to scale up the seed observation data while keeping the temporal and spatial correlations of the seed data.

*Temporal Scaling.* Data generated by sensors typically follows a pattern over time (e.g., a temperature reading may gradually increase, then decrease as the day progresses). Patterns repeat over days, and sensor data may be generated using several patterns – daily patterns, seasonal patterns, etc. We first detect patterns in the seed data (at different time periods) and then repeat it to generate synthetic data at the desired scale. Controlled noise is added to the data to capture the variability of the real world.

*Speed Scaling.* To increase data velocity, we synthetically increase the sensor data generation frequency by exploiting temporal locality in sensor data values. For example, if a temperature sensor reads 70 F degrees at time $t$, and 71 F at $t + \delta t$, we extrapolate temperature readings (based on the speed scaling factor specified in the configuration) by randomly selecting a temperature value from the range [70-$\Delta_{temp}$, 71 + $\Delta_{temp}$] where $\Delta_{temp}$ is controlled noise used to capture variation in real sensor values. $\Delta_{temp}$, itself, is calculated based on prior sensor readings in the seed file.

*Device Scaling.* To generate data from synthetic sensors (added during metadata scaling), we exploit spatial locality in sensor readings. For example, a temperature sensor in one location will likely generate readings not too different than those from another temperature sensor in a nearby location. To generate sensor readings from synthetic sensors, the data generator determines readings of nearby real sensors in the seed data and assigns noise-added copies of its readings as

[2]The complete list of configuration parameters supported is available in Appendix C

the readings for the new sensor.

*User Scaling* To generate data corresponding to synthetic users (added during metadata scaling), we generate synthetic trajectories of people based on transition probabilities between neighboring regions. The high-level trajectory data is then converted into sensor data by determining relevant sensors that could observe the user based on his/her trajectory (by using the inverse coverage function).

**Query Generation.** In addition to generating data, the tool also generates an actual query workload based on the query templates described in the previous section. The input to the query generator 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 $\delta_a$ and $\delta_b$ are provided in the configuration file. List based query parameters, for example 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 values for $n_a$ 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.

# 4. DATABASE SYSTEMS AND MAPPINGS

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 [4] website; 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. GriDB, 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++[31] (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).

**Other database systems.** We also chose *CrateDB*,

| 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

Table 1: Different DBMS and their capabilities.

which is advertised as a SQL database for time series 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 [23] and therefore naturally supports inverted indexes, although it has no support for B-tree based indices.

## 4.1 Model Mapping

There are several ways in which the schema in Section 3.2 can be represented in the underlying database systems. We explore multiple mappings of IoT data to the underlying databases, mappings that are broadly characterized based on how sensor data is stored in the database: single table for observations from all sensors (A) mapping; multiple tables, with one per sensor type (T) mapping; and a multiple tables, with one per sensor instance (S) mapping.

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 2 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.

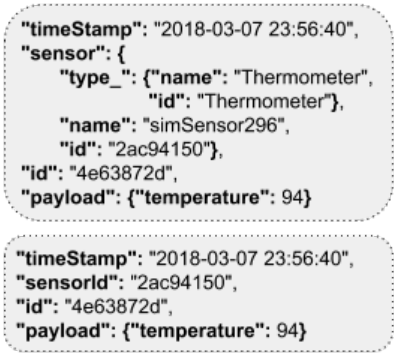



Figure 2: Mappings for document stores (A1 and A2).

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 Observations 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 the PostgreSQL database whenever it is required by the application.

## 4.2 Expected Performance For IoT Workloads

Table 2 lists our apriori expectations regarding the impact of the database technology choice on the performance of different queries. The last column 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 the SmartBench data model is mapped to the underlying data model supported by the database system, and also the ease of use of the system.

Systems with row based storage should perform better compared to column stores for insertion. In addition, systems based on LSM/TSM trees should support insertions even more efficiently. Based on the above, we would expect AsterixDB and InfluxDB to perform the best on insertions.

Simple metadata queries (Q1-Q2) are on small tables, so every database should perform well on them. Queries selecting observations, based on timestamps (Q3-Q5), and queries aggregating statistics (Q6) should benefit both from the presence of secondary indices on time as well as column store

| 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 |

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

technology. In addition, queries involving temporal predicates/operators (Q3-Q6) should benefit from the specialized storage offered by timeseries databases. Thus, we expect InfluxDB and GridDB to perform well for such queries, especially when the selectivity of the condition on the timestamp is low. We would expect systems such as AstetixDB and PostgresSQL, which support efficient implementations of joins and advanced query optimization, to perform better on complex queries involving large joins (Q7-Q10). We also expect SparkSQL to perform better on these complex queries due to Parquet files based columnar storage.

In terms of scaling out to a multi-node setup, two key factors include (a) horizontal sharding capability, and (b) efficiency of a distributed query processing engine. Since AsterixDB and SparkSQL have the most advanced distributed query processing engines of the ones tested on the benchmark, we expect them to perform the best especially for complex queries. GridDB, InfluxDB and MongoDB also include mechanisms for data sharding (implicit sharding on sensor-id in GridDB and on (sensor-id, timerange) in InfluxDB), but lack an optimized distributed query engine. Hence, we expect them to benefit from their scale out on Q3-Q5, but not on complex queries requiring distributed joins and aggregations such as Q7-Q9.

# 5. EXPERIMENTS AND RESULTS

| Dataset | Single Node | | Multi Node | |
|---|---|---|---|---|
| | Small | Large | Small | Large |
| Users | 1000 | 2500 | 10000 | 25000 |
| Sensors | 300 | 600 | 3000 | 15000 |
| Rooms | 300 | 600 | 2500 | 7500 |
| Days | 90 | 120 | 150 | 180 |
| Frequency | 1/300s | 1/200s | 1/150s | 1/100s |
| Observations | 14 mil | 30 mil | 150 mil | 800 mil |
| Semantic Observations | 14 mil | 30 mil | 150 mil | 800 mil |
| Size | 12GB | 25GB | 125GB | 600GB |

Table 3: Dataset sizes.

**Dataset and queries.** The datasets were generated by the data and query generation tool presented in Section 3.4. As the seed data for the tool, we used real data from an IoT data management system, TIPPERS [10], deployed in Donald Bren Hall (DBH) at the University of California, Irvine (UCI). DBH is a 90000+ square feet 6-story building with 125 faculty offices, 90 research labs, multiple classrooms, several lecture halls, and departmental offices. DBH is equipped with various kinds of sensors including 116 HVAC data points that provide readings from different equipment (e.g., vents and chillers), 216 thermometers measuring temperature in different regions of the building, 40 surveillance cameras covering all the corridors and doors (for security purposes), 64 WiFi access points (APs) to provide inhabitants with Internet connectivity, 200 beacons covering the public areas (e.g., lounges, kitchens), doors to access the building, and a number of meeting rooms, labs, and offices; there are also 50 outlet meters that measure the energy usage of shared equipment (e.g., projectors, printers, refrigerators) and computers of members of the ISG research group. The TIPPERS instance has been running for almost two years and has collected over 200 million observations from these sensors. Also, TIPPERS generates higher-level semantic information through virtual sensors (see Section 3.2) by running machine learning algorithms (mainly related to the presence of the inhabitants in the different rooms of the building and the occupancy of such rooms).

We used as a seed the data collected by TIPPERS for a period of one week from 340 rooms (divided into 16 types) using three types of sensors – WiFi Access Points (64 units), plug meters (20 units), and thermometers (80 units). This data also includes location and occupancy data generated by TIPPERS using various virtual sensors. Using this seed data, the data generation tool was used to generate three different sizes of datasets (see Table 3) to evaluate database systems over SmartBench. The query parameters were also generated by the SmartBench generation tool. We controlled the query selectivity by controlling the range of time period $t_e - t_b$ specified in the queries. For base experiments, we restricted the time range to be $1\ day < t_e - t_b < 4\ days$. The impact of query selectivity on the performance of different systems was explored by generating queries with larger time ranges, as we will explain later. A complete list of the query generation parameters used can be found in the Appendix C.

**Database System Configuration.** For the experiments, we used MongoDB CE v3.4.9, GridDB SE v3.0.1, AsterixDB v0.9.4, PostgreSQL v9.6.5, CrateDB v2.1.6, InfluxDB v1.7 and SparkSQL v2.1.6. 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 for the databases tested, setting the buffer cache size to 2 GB per node for all. Also, for all of the database systems we created a secondary index on the timestamp attribute, 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. Furthermore, 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. The complete configuration used for each database system can be found in Appendix A.

## 5.1 Single Node Experiments

The first two datasets in Table 3 are used for the experiments on a single node with an Intel i5 CPU 8GB RAM, 64 bit Ubuntu 16.4, and 500 GB HDD machine.

| Dataset | Mapping | Single Node | | Multi Node |
|---|---|---|---|---|
| | | Small | Large | Small |
| griddb | S | 4 | 8.6 | 85 |
| | T | 5.6 | 11 | NA |
| postgresql | A | 8 | 18 | NA |
| | T | 7.5 | 17 | NA |
| mongodb | A1 | 7 | 16 | NA |
| | A2 | 6.2 | 15 | 115 |
| asterixdb | A1 | 9 | 22 | NA |
| | A2 | 7.5 | 20 | 128 |
| cratedb | A | 9 | 19 | NA |
| | T | 12 | 30 | 140 |
| sparksql | A | 7.5 | 14 | NA |
| | T | 6.5 | 12 | 95 |
| influxdb | S | 2.8 | 6.8 | NA |

Table 4: Dataset sizes after ingestion including indices (GB).

| Dataset | Mapping | Single Node | | Three Nodes |
|---|---|---|---|---|
| | | Small | Large | Small |
| griddb | S | 66.44 | 158.37 | 8941.27 |
| | T | 84.89 | 188.98 | NA |
| postgresql | A | 1929.21 | 2974.75 | NA |
| | T | 510.85 | 836.34 | NA |
| cratedb | A | 588.81 | 1459.86 | NA |
| | T | 789.03 | 1938.08 | 54750.11 |
| mongodb | A1 | 1359.02 | 4837.34 | NA |
| | A2 | 685.07 | 2413.93 | 87302.95 |
| asterixdb | A1 | 1384.43 | 2895.18 | NA |
| | A2 | 673.26 | 1268.96 | 9817.87 |
| influxdb | T | 44.24 | 101.37 | NA |

Table 5: 10% Insert runtimes(seconds).

**Experiment 1 (Insert Performance Test):** In this experiment, we compare the insert performance of different databases 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 5000 rows/documents which amounts to 5 minutes worth of observation data for the large dataset. WAL is enabled for all the database 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 5 shows the results for this experiment for both the small and large datasets (columns "Single Node (Small)" and "Single Node (Large)", respectively).

As expected, InfluxDB performs best due to its TSM based storage engine and also due to its support for columnar compression which reduces the size of data inserted to about one-third (Table 4). Interestingly, GridDB (with mapping S) performs the second best. Like InfluxDB, the size of data stored due to insertions in GridDB is relatively small (compared to other row stores); see Table 4. Also, GridDB 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. In addition, GridDB 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 (viz., two indices versus one, larger indices for data from all sensors of a given type versus an index over single sensor data).

Between MongoDB and AsterixDB, AsterixDB performs better due to its support for LSM storage even though MongoDB has a slightly smaller database size from insertions due to its row-level compression. Their performance gap increases with increased number of documents when data starts to spill to the disk. AsterixDB's performance, which supports LSM, is less affected, while MongoDB experiences increased I/O overheads.

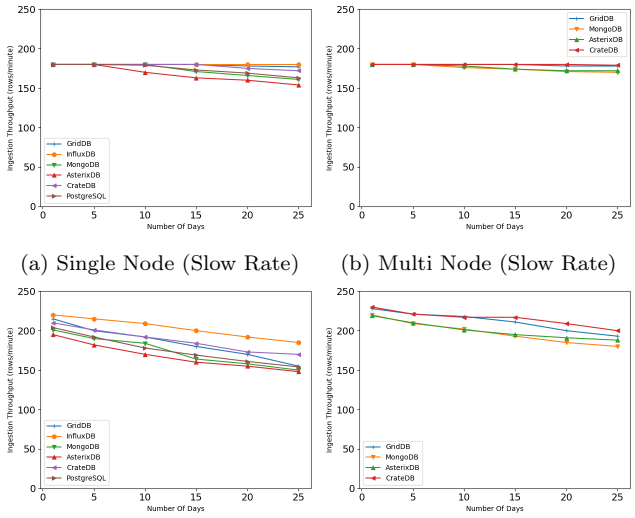
PostgreSQL, however, initially surprisingly, performs better compared to AsterixDB even though it lacks LSM storage. PostgreSQL supports heap storage and new records are inserted in memory with the data spilling over to disk only when the memory is full. Updates to index pages, however, can result in random I/O while in AsterixDB that would not be the case (since its indexes are also LSM trees). The reasons for PostgreSQL's superior performance are: (a) The sequentially written heap storage mitigates many of the advantages of LSM for an insert-only workload; (b) 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; (c) LSM storage had a processing overhead due to several merges which was not the case for the heap storage; and (d) The overall size of records stored in PosgreSQL was smaller compared to the records in AsterixDB (which is a document store).

Finally, CrateDB performed the worst, as expected given that it has a columnar storage engine with no compression (by default). Also, since it does not allow selections on columns with no indexes, we created indexes on multiple columns, causing more index updates at insertion time. Data ingestion takes about 20% more time on mapping T compared to mapping A for CrateDB (Table 5), since this mapping involves more columns that need to be indexed.

In addition to insert, we also conducted an experiment on insert with enrichment. The IE pipeline (described in Section 3.3 consisted of a 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 3.4, with min and max time interval of 4 and 8 hours respectively. We chose $\tau$ (busy wait to simulate execution of enrichment function) to be 100ms, which is similar to the time taken for enrichment functions in [39] 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 $\tau$, enrichment cost quickly dominate and becomes a bottleneck and the rate at which semantic observations are generated (10,000/second) could easily be sustained by all the database 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 [39] 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 DB system.

**Experiment 2 (Query Performance Test):** The second experiment runs the benchmark queries described in Section 3.3. For all systems, except GridDB and InfluxDB, every benchmark query maps to a single query in the query language supported by the database systems. 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. Our implementation pushes selections down, uses the best join order, and is equivalent to an index nested loop based join. 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. Appendix D shows the complete implementations of the

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




(a) Single Node (Slow Rate) (b) Multi Node (Slow Rate)

(c) Single Node (Fast Rate) (d) Multi Node (Fast Rate)

Figure 3: Insertion throughput with virtual sensors.

queries for GridDB, InfluxDB, and the other database systems. Figure 5 presents the query performance results for the large dataset.

*Metadata Queries (Q1,Q2).* These queries were mostly included to compare the functionality aspect (ability to store arbitrary data) of the database systems and, as expected, all databases performed relatively well on both these queries except InfluxDB (a specialized timeseries database system, with its tags and value based model) since it does not provide a way to store complex metadata.

*Simple Selection and Roll Up (Q3-Q6).* As expected, time series databases performed very well on these queries. Q3-Q5 are range selection queries over timestamp. GridDB (mapping S) performs very well since it stores data clustered based on timestamp (the primary key). InfluxDB's performance is comparable (slightly better) compared to GridDB. The performance of PostgreSQL and CrateDB are roughly similar since these queries required most of the columns to be retrieved and did not include any aggregation operations (therefore the columnar storage of CrateDB did not provide much benefit). MongoDB's and AsterixDB's performance is comparatively slower since these queries involve scanning a set of rows and, hence, their larger record sizes (due to their document model) slows down their overall query performance. Also, the LSM tree based data storage used in AsterixDB 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 on these simpler queries due to the data compression supported by its WiredTiger storage engine and the use of index compression by default that helps in better secondary index scans.

*Complex Queries (Q7-Q10).* For these queries, we expected GridDB (with either mapping A or T) and InfluxDB not to perform well, since these queries require complex joins and aggregations, and neither of the systems support JOIN or GROUP BY operators. As these operators are implemented in the benchmark code, we expected that in general application-level operators would not perform as well native database operator implementations. However, at low selectivity (date range of only 1 to 4 days) in the base experiments, the number of rows to be joined and grouped after all the selection predicates are executed is small and so application level joins did not actually cause significant overhead. In fact, except for queries Q7 and Q8, GridDB and InfluxDB outperformed the other databases (with native JOIN support) other than PostgreSQL and CrateDB[4]. However, both GridDB and InfluxDB resulted in an out of memory error for Q9 since the application level sort-based GROUP BY operator could not store the number of rows fetched in memory. GridDB's performance drops down significantly with mapping T since it suffers from the drawback of not being able to use the primary index on any of the queries, while InfluxDB performs slightly better compared to GridDB due to its faster read performance (see Q3-Q5).

CrateDB, being 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 queries Q7, Q9. However interestingly, it took a considerable amount of time on query 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 query Q10 as the query involved most of the columns from the table.

All the queries on MongoDB perform better with mapping A2 compared to A1. This observation 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 very 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 has better JOIN support compared to MongoDB. Therefore, on queries Q7-Q10, where MongoDB either timed out or took a long time to finish, AsterixDB performed much better. However, its performance on these queries were not as good as in PostgreSQL because AsterixDB for queries Q3-Q5 has to scan rows which are comparatively larger in size and PostgreSQL was able to come up with a better query plan since it has a more mature optimizer and it stores statistics about the data, which AsterixDB does not. SparkSQL, even with columnar storage (parquet files on HDFS), did not perform well on queries Q7-Q10 since it has to scan the entire dataset for all the queries due to the lack of support for secondary indexes.

**CPU and IO Usage (Figure 6):** 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 [**?**]). 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

[4]In Experiment 3, we will compare application-level and database level joins for different levels of selectivity to gain better insight into the tradeoffs.

| Database | M | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 | Q10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| griddb/* | S | 0.002 | 0.002 | 0.006 | 0.006 | 0.12 | 0.013 | 0.048 | 2.24 | 0.019 | 0.60 |
| | T | 0.001 | 0.011 | 8.13 | 15.06 | 2.48 | 9.66 | 37.14 | 22.48 | 0.41 | 8.37 |
| postgresql | A | 0.001 | 0.021 | 0.98 | 1.15 | 1.01 | 0.98 | 24.81 | 1.86 | 2.19 | 1.35 |
| | T | 0.001 | 0.008 | 0.31 | 1.19 | 0.432 | 0.973 | 1.01 | 0.36 | 0.31 | 0.24 |
| cratedb | A | 0.006 | 0.19 | 0.41 | 1.07 | 61.26 | 0.13 | 0.68 | TO | - | - |
| | T | 0.004 | 0.27 | 1.41 | 0.47 | 1.59 | 0.19 | 0.79 | TO | - | - |
| mongodb | A1 | 0.001 | 0.002 | 0.61 | 0.72 | 8.83 | 0.53 | TO | TO | 5.09 | 15.59 |
| | A2 | 0.001 | 0.002 | 0.20 | 0.35 | 2.03 | 0.24 | TO | TO | 3.46 | 5.83 |
| asterixdb | A1 | 0.02 | 0.17 | 2.83 | 3.10 | 4.20 | 3.19 | 30.91 | 23.39 | 2.342 | 2.235 |
| | A2 | 0.02 | 0.04 | 1.84 | 2.01 | 3.38 | 1.67 | 12.65 | 10.33 | 0.86 | 1.55 |
| sparksql | A | 0.005 | 0.01 | 1.53 | 1.49 | 0.32 | 0.89 | 50.32 | 34.12 | 1.45 | 6.87 |
| | T | 0.006 | 0.01 | 1.87 | 1.24 | 0.34 | 0.98 | 13.22 | 10.12 | 1.87 | 3.45 |
| influxdb | T | - | - | 0.005 | 0.09 | 0.025 | 0.073 | 0.15 | 2.52 | 0.10 | 1.08 |

Table 6: Query runtime (seconds) on small dataset (single node, 5 minutes timeout).


Figure 4: Performance of App vs DB joins.

| Database | M | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 | Q10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| griddb/* | S | 0.001 | 0.003 | 0.32 | 0.95 | 1.56 | 1.02 | 48.94* | 36.98* | OOM | 4.35 |
| | T | 0.001 | 0.009 | 50.83 | 75.15 | 9.75 | 34.02 | 230.13* | 142.39* | OOM | 34.51 |
| postgresql | A | 0.001 | 0.024 | 5.76 | 6.08 | 6.51 | 4.78 | 103.62 | 25.87 | 43.31 | 8.34 |
| | T | 0.001 | 0.016 | 2.96 | 4.63 | 1.27 | 6.14 | 37.23 | 12.16 | 10.96 | 7.95 |
| cratedb | A | 0.008 | 0.19 | 2.42 | 6.47 | 15.61 | 1.04 | 3.47 | 265.45 | 2.82 | 18.61 |
| | T | 0.004 | 0.31 | 3.92 | 2.91 | 4.85 | 0.79 | 1.98 | 198.45 | 1.56 | 15.91 |
| mongodb | A1 | 0.002 | 0.017 | 8.97 | 10.86 | 39.54 | 5.48 | TO | TO | 24.49 | 29.93 |
| | A2 | 0.002 | 0.015 | 7.92 | 8.98 | 15.03 | 4.23 | TO | TO | 19.06 | 24.14 |
| asterixdb | A1 | 0.024** | 0.18 | 12.59 | 13.84 | 20.84 | 11.19 | 128.20 | 91.28 | 14.93 | 10.39 |
| | A2 | 0.023** | 0.073 | 9.81 | 11.87 | 16.91 | 10.98 | 84.76 | 80.564 | 4.97 | 7.68 |
| sparksql | A | 0.007 | 0.012 | 17.98 | 21.87 | 30.84 | 20.48 | 158.47 | 183.09 | 8.04 | 25.69 |
| | T | 0.007 | 0.011 | 15.96 | 18.65 | 24.31 | 17.63 | 110.58 | 148.36 | 7.89 | 15.03 |
| influxdb | T | - | - | 0.15 | 0.70 | 1.28 | 0.58 | 25.74 | 30.65 | OOM | 10.19 |

*GridDB + application-level joins. **AsterixDB has a lower bound of 10-20ms to do any query due to its current operation mode (which is not optimized for simple or non-cluster queries)

Figure 5: Query runtime (seconds) on large dataset (single node, 15 minutes timeout).

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 unsuported 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 [**?**], 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.

**Experiment 3 (Application Level vs Database Joins):** Experiment 2 showed that application-level joins implemented over systems such as GridDB and InfluxDB (which do not support a native join implementation) did not cause applications to incur much performance impact. In this experiment, we further study the performance of such joins with respect to query selectivity levels. In particular, we implemented Q8 (which requires a self join of the presence table, with 15 million rows) in PostgreSQL in a second way - using an application-level nested loop join[5]. At low selectivity, for the native join case, PostgreSQL's optimizer selected a plan consisting of an index scan on the outer presence table to filter based on the timestamp predicate followed by a nested loop index join with the inner presence table. For higher selectivities it changed the plan to a sequential scan of the outer presence table followed by a nested loop index join. To implement application level joins, we first sent a selection query to the outer presence table, and then, for each row in the result set, sent a selection to the inner presence table. PostgreSQL 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.

Figure 4 shows the results for this test. As expected, database-level joins outperform application-level joins. This is due to the higher overhead of the latter (e.g., multiple independent queries compiled separately). Interestingly, however the results show that if the selectivity is small enough, in our results for this specific query below 0.5 percent of the dataset, application-level joins perform very competitively to a system level join. This is interesting since in IoT applications, joins are often between small metadata tables and large timeseries data. Furthermore, 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 work well with application-side join code on queries with joins as well.

**Experiment 4 (Effect of time ranges):** Query performance, as well as, the relative performance of different databases, depends upon query selectivity. In SmartBench the key factor controlling query selectivity is time ranges. Indeed, queries in a real context will involve different time ranges based on the application. For example, a building management application may wish to compute aggregated occupancy levels over an interval of several months, while another application to monitor the interactions of a person and the space may retrieve data for a single day. To understand the effect of selectivity, we chose two queries – Q6 and Q8 – and compared all the systems while varying the associated time ranges (one day, one week, two weeks, one month, and one months) for this experiment.

Figure 8a shows the performance of different systems with increasing selectivity on Q6 (simple selection followed by

[5]We used the same system (PostgreSQL) to compare an application-level versus a native join to remove effects of other performance factors which using different database systems for the experiments would have introduced.

| Query | griddb | | | | | cratedb | | | | | mongodb | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 3 | 6 | 9 | 12 | 1 | 3 | 6 | 9 | 12 | 1 | 3 | 6 | 9 | 12 |
| Q1 | 0.01 | 0.012 | 0.015 | 0.015 | 0.02 | 0.03 | 0.015 | 0.010 | 0.010 | 0.01 | 0.25 | 0.05 | 0.01 | 0.01 | 0.01 |
| Q2 | 0.02 | 0.027 | 0.047 | 0.04 | 0.04 | 0.31 | 0.22 | 0.20 | 0.20 | 0.21 | 0.13 | 0.05 | 0.02 | 0.02 | 0.02 |
| Q3 | 5.25 | 3.95 | 3.74 | 2.53 | 1.81 | 55.87 | 35.72 | 26.76 | 18.39 | 13.78 | 211.44 | 127.40 | 87.29 | 75.52 | 69.75 |
| Q4 | 22.67 | 18.72 | 20.82 | 32.67 | 53.91 | 32.43 | 24.09 | 18.44 | 14.34 | 11.72 | 1614.06 | 375.93 | 98.67 | 129.32 | 171.53 |
| Q5 | 59.66 | 60.37 | 70.37 | 81.76 | 100.83 | 217.65 | 110.42 | 64.52 | 33.85 | 18.98 | 121.55 | 99.22 | 124.40 | 262.37 | 569.46 |
| Q6 | 32.19 | 31.94 | 43.12 | 40.65 | 39.09 | 65.82 | 29.95 | 15.63 | 14.75 | 14.17 | 306.97 | 211.41 | 187.34 | 165.20 | 161.96 |
| Q7 | 1053.01 | 902.59 | 824.23 | 840.33 | 875.32 | 45.41 | 32.05 | 23.44 | 16.91 | 13.78 | TO | NA | NA | NA | NA |
| Q8 | 997.34 | 853.62 | 865.13 | 947.65 | 1049.81 | TO | TO | TO | TO | TO | TO | NA | NA | NA | NA |
| Q9 | OOM | OOM | OOM | OOM | OOM | 8.93 | 5.21 | 3.51 | 3.14 | 2.97 | 74.20 | 45.15 | 33.57 | 12.57 | 4.74 |
| Q10 | 154.79 | 118.66 | 130.98 | 126.76 | 131.22 | 115.19 | 74.64 | 54.35 | 30.82 | 19.14 | 476.82 | 541.25 | 614.54 | 664.83 | 759.18 |

| Query | asterixdb | | | | | sparksql | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 3 | 6 | 9 | 12 | 1 | 3 | 6 | 9 | 12 |
| Q1 | 0.15 | 0.098 | 0.07 | 0.06 | 0.05 | 0.03 | 0.02 | 0.02 | 0.02 | 0.02 |
| Q2 | 0.41 | 0.30 | 0.22 | 0.25 | 0.30 | 0.03 | 0.024 | 0.026 | 0.026 | 0.03 |
| Q3 | 178.42 | 95.86 | 65.32 | 26.58 | 12.79 | 131.06 | 105.47 | 85.42 | 70.65 | 61.06 |
| Q4 | 198.13 | 104.80 | 66.06 | 25.16 | 10.92 | 304.41 | 222.91 | 176.17 | 125.87 | 94.32 |
| Q5 | 181.45 | 95.70 | 63.18 | 21.75 | 9.21 | 411.48 | 327.09 | 280.18 | 200.90 | 155.08 |
| Q6 | 211.35 | 101.01 | 62.68 | 24.72 | 10.03 | 371.86 | 283.62 | 223.89 | 156.75 | 115.15 |
| Q7 | 1243.59 | 548.65 | 308.23 | 200.76 | 135.41 | 2620.19 | 1910.58 | 1402.12 | 928.0 | 700.23 |
| Q8 | 980.51 | 490.46 | 252.05 | 163.75 | 108.75 | 3244.67 | 1868.46 | 1329.90 | 880.50 | 646.91 |
| Q9 | 321.86 | 156.21 | 81.598 | 53.28 | 35.84 | 282.05 | 160.49 | 120.18 | 95.84 | 78.98 |
| Q10 | 312.76 | 140.07 | 79.334 | 39.16 | 22.95 | 225.27 | 195.39 | 172.90 | 150.50 | 136.87 |

Table 7: Query runtimes (s) on mulit node, small dataset setup(1 hour timeout).

aggregation on time). While the runtimes for GridDB, InfluxDB, and CrateDB increase with increasing selectivity, these systems continue to perform better compared to other database systems. PostgreSQL performed as well as the timeseries database systems for low selectivities, 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 gets worse with increased selectivity due to their comparatively larger record sizes. Figure 8b shows the performance of different database systems on Q8 (join of two large tables followed by aggregation). AsterixDB, SparkSQL, and CrateDB all 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 performed well on low values of selectivity. However, with increase in selectivity the performance of PostgreSQL dropped. For timeseries databases, as mentioned before, we wrote index nested loop based application joins which performed quite well for low selectivity values, but their performance grew super-linearly with increasing selectivity.

## 5.2 Multi-Node Experiments

For multi-node experiments we used larger datasets (see Table 3). Data was partitioned over 1, 3, 6 9and 12 nodes (each node is an Intel i5 CPU, 8GB RAM, 800GB HDD, and CentOS 7 machine). Nodes are connected via a 1 Gbps network link. The database system 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 5 (Insert Performance Test):** Table 5 shows the results for the insertion test for the 3 node experiment (right most column). Similar to single node experiments, GridDB again performs well on inserts as expected, although notice that the per tuple insertion time has increased from 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 increases 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 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 increases, 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 6 (Query Performance Test):** Table 7 shows the results for the execution of different queries on the 1, 3, 6, 9 and 12 node configurations, while Table 8 shows the results for a 12 node configuration for the large database with 1.6 billion rows [6]

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. However, 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 [7]. For other queries, that require data to be

[6]Note we do not include results for GridDB on the 12 node configuration since GridDB does not support bulk insertion and at its insertion rates (10K tuples per second), insertion time for the large database would take over 50 hours.

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

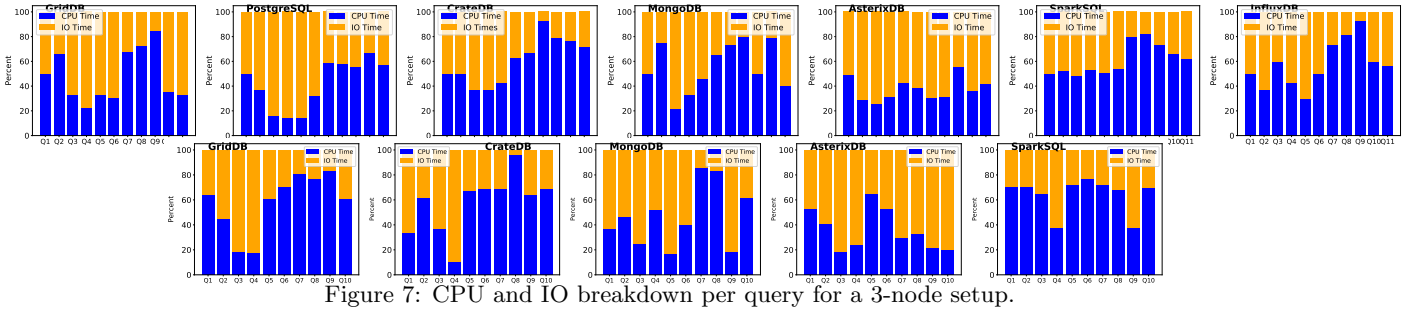


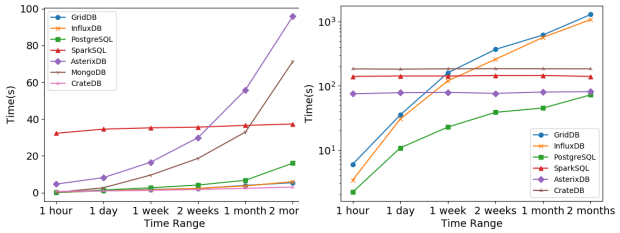

Figure 7: CPU and IO breakdown per query for a 3-node setup.

fetched from multiple containers, GridDB performance does not improve since GridDB 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 q sequential program that GridDB was not able to leverage any parallelism from the multi-node setup. We, then, implemented multithreaded programs to fetch data from multiple containers simultaneously for GridDB. 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. However, we did not see much improvement on queries Q7 and Q8.

In case of 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 [17]. The Cluster controller reads the sharding information and other metadata from a special node 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 however, 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. However, even with the mongos service, MongoDB does not support joins between two sharded collections, so we skipped queries Q7 and Q8 for multi-node experiments. Also in MongoDB, all unsharded collections are stored 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




(a) Q6 (b) Q8

Figure 8: Performance with time selectivity.

an increasing number of nodes, as its optimizer started to pick a collection scan over an index scan.

## 5.3 Mixed Workloads Experiments

The benchmark supports also mixed workloads where insertions and queries are run together. In the following we compare system performance under such online mixed workload. In this experiment, queries of the same template are executed in parallel with the data ingestion. We used two different levels of data insertion rate: slow and fast. For the slow rate, the data is available to insert at the rate of 10,000 observations per second, whereas for the fast the rate is 50,000 observations per second. The experiment was repeated multiple times with different sizes of data (based on days) already ingested.

**Experiment 8 (Online inserts and queries):** Figure 9, 10, 11, 12 shows the average query latency for each query w.r.t. the number of days (the workload is starting from a point where data for a fixed number of days is already inserted) for both slow and fast 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. However, this rate of increase in latency is comparatively higher in case of faster insert rate. This is expected since the queries are now running in parallel with a much higher load of inserts. The results for multinode version of the experiment (DBs running on a cluster of 3 servers) shows a similar scalabilitiy trend for the DBs 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.

**Experiment 9 (Continuous Query - Figure 13):** 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 the occupancy data of all the rooms in the last 10 seconds time window, followed by running a selection query on the database system to discover the rooms of type "Lecture Hall" and finally joined it with the buffered data. Figure 13 shows the results for slow and fast data generation rate on a single node as well as a multi-node (3 node)

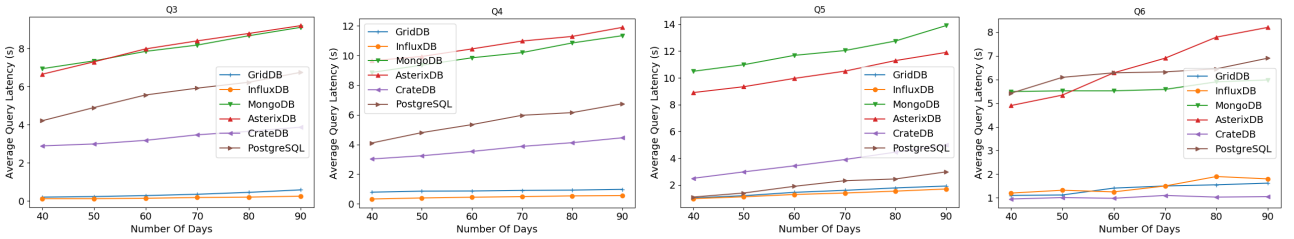



Figure 9: Query latency (Single Node Slow Rate).

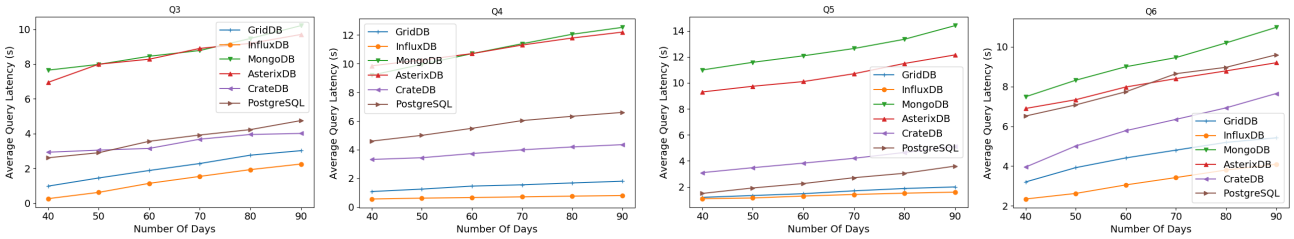



Figure 10: Query latency (Single Node Fast Rate).

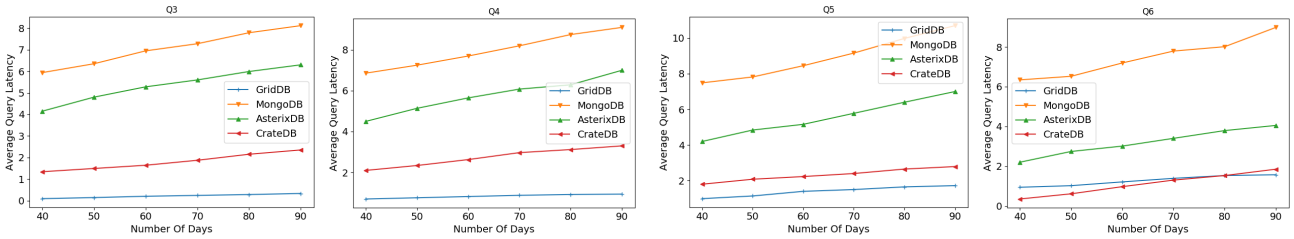



Figure 11: Query latency (Multi Node Slow Rate).

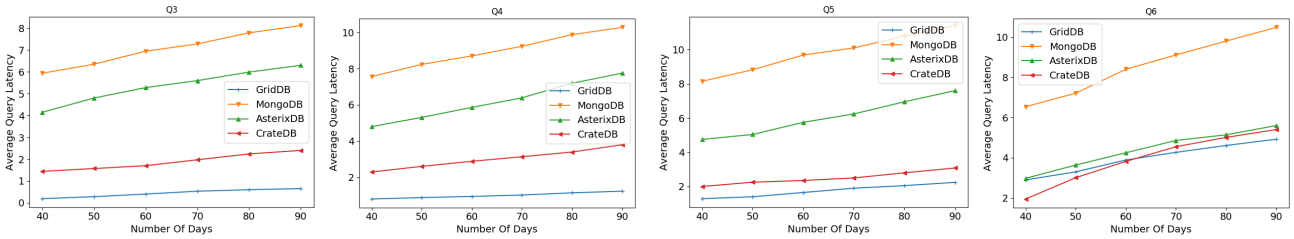
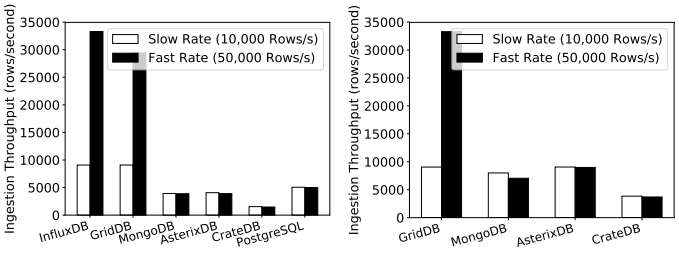



Figure 12: Query latency (Multi Node Fast Rate).




(a) Single Node (b) Multi Node

Figure 13: Insertion throughput (CQ)

| 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 | 1245.57 |
| Q8 | TO | NA | 876.44 | 1197.45 |
| Q9 | 12.18 | 30.46 | 162.13 | 180.77 |
| Q10 | 103.23 | 2235.8 | 80.73 | 251.35 |

Table 8: Query runtimes (s) on large dataset (12 nodes).

setup. Since the query is executed as part of the application, the difference in performance of different DBMSs can be attributed to different ingestion rate they support 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 relative performance of systems leads to several interesting observations for IoT data management which are; 1) IoT data, in addition to consisting of time varying sensor observations, also consists of additional information such as spatial relationships, entities, events, etc. Applications consist of queries over all such data. 2) Several ways exist to represent heterogeneous sensor data in underlying databases. These mappings affect insertion and query performance in some cases significantly. E.g., our experiments show that normalized representations perform better compared to more nested structures for both insertion and queries). 3) IoT query workloads range from selections on temporal attributes to queries requiring multiway-joins, grouping, and aggregations. Often, these joins can be simple - for instance, interpreting semantically meaningful observations from raw data may require joins of (large) sensor data with (small) metadata. 4) An IoT system must support a high rate of data arrival (velocity) and thus needs to support fast insertions. However, its data is mostly append-only with rare updates (metadata). Further, data volume over time can be large, so scalability (scale up as well as scale out) is important.

We next highlight some key observations about the suitability of different database systems as well as implementation choices for IoT workloads.

• While specialized timeseries databases are suited for sensor data (fast insertion and time-based selection queries), they do not provide natural ways to store the other information (e.g., spatial relationships, entities, events) needed for building IoT applications. 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 in for InfluxDB, where we used PostgreSQL for storing non-timeseries data).

• While document stores are suited to represent heterogeneous data, 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 true 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 [9].

• Time series databases, such as InfluxDB and GridDB, performed well on inserts, and simple selection queries as well as on 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 execute 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 ingest. Such an architecture can be sub-optimal specially if complex enrichment function need to run queries to retrieve past data [39]. 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 several other other domains, while different systems offer different advantages, there is no single system that offers the "best" choice. The emerging field of Polystores [21], which aims to provide integration middleware allowing applications to store different parts of their data in different underlying databases, may be a relevant solution.

# 7. REFERENCES

[1] Apache Kafka, Stream Processing Engine. https://kafka.apache.org/intro. [Online; accessed Feb-2018].

[2] Apache Storm, Stream Processing Engine. http://storm.apache.org/about/integrates.html. [Online; accessed Feb-2018].

[3] Couchbase NoSQL Database. https://www.couchbase.com/. [Online; accessed Feb-2018].

[4] DB-Engines Ranking. https://db-engines.com/en/ranking. [Online; accessed Feb-2018].

[5] GridDB, NoSQL Database System For IoT. https://griddb.net/en/. [Online; accessed Feb-2018].

[6] InfluxDB, Timeseries Database System. https://www.influxdata.com/_resources/. [Online; accessed Feb-2018].

[7] MongoDB. https://www.mongodb.com/. [Online; accessed Feb-2018].

[8] PostgreSQL, Relational Data Management System. https://www.postgresql.org/. [Online; accessed Feb-2018].

[9] Teradata. http://www.teradata.com. [Online; accessed Feb-2018].

[10] TIPPERS. http://tippersweb.ics.uci.edu/. [Online; accessed Feb-2018].

[11] TPC-C Bechmark. http://www.tpc.org/tpcc/. [Online; accessed Feb-2018].

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

[13] 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. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014.

[14] 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 *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491. VLDB Endowment, 2004.

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

[16] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[17] 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 International Conference on Data Engineering*, pages 1151–1162. IEEE, 2011.

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

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

[20] 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.

[21] 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.

[22] 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 *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 1197–1208. ACM, 2013.

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

[24] 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.

[25] 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 *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 101–112. IEEE, 2015.

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

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

[28] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 69–78. IEEE, 2014.

[29] 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.

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

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

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

[33] 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 International Conference on Data Engineering (ICDE)*, pages 1519–1530. IEEE, 2018.

[34] 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.

[35] 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.

[36] 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.

[37] 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. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[38] 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 *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499. IEEE, 2014.

[39] X. Wang and M. Carey. An idea: An ingestion framework for data enrichment in asterixdb. *Proceedings of the VLDB Endowment*, 12(11):1485–1498, 2019.

# APPENDIX

# A. DATABASE SYSTEMS CONFIGURATIONS

## A.1 AsterixDB

```
[nc/red]
txn.log.dir=data/red/txnlog
core.dump.dir=data/red/coredump
iodevices=data/red

[nc/blue]
ncservice.port=9091
txn.log.dir=data/blue/txnlog
core.dump.dir=data/blue/coredump
iodevices=data/blue

[nc]
storage.subdir=storage
address=127.0.0.1
command=asterixnc
storage.buffercache.size = 2147483648

[cc]
address = 127.0.0.1

[common]
log.level=INFO
```

## A.2 MongoDB

```
storage:
  wiredTiger:
     engineConfig:
        cacheSizeGB: 2
```

## A.3 PostgreSQL

```
** updated part
# - Memory -

shared_buffers = 2048MB          # min 128kB
              # (change requires restart)
```

## A.4 InfluxDB

```
cache-max-memory-size = 2g
```

## A.5 GridDB

```
{
  "dataStore":{
     "dbPath":"data",
     "storeMemoryLimit":"2048MB",
     "storeWarmStart":true,
     "concurrency":1,
     "logWriteMode":1,
     "persistencyMode":"NORMAL",
     "affinityGroupSize":4
  },
  "checkpoint":{
     "checkpointInterval":"1200s",
     "checkpointMemoryLimit":"1024MB",
     "useParallelMode":false
  },
  "cluster":{
     "servicePort":10010
  },
  "sync":{
     "servicePort":10020
  },
  "system":{
     "servicePort":10040,
     "eventLogPath":"log"
  },
  "transaction":{
     "servicePort":10001,
     "connectionLimit":5000
  },
  "trace":{
     "default":"LEVEL_ERROR",
     "dataStore":"LEVEL_ERROR",
     "collection":"LEVEL_ERROR",
     "timeSeries":"LEVEL_ERROR",
     "chunkManager":"LEVEL_ERROR",
     "objectManager":"LEVEL_ERROR",
     "checkpointFile":"LEVEL_ERROR",
     "checkpointService":"LEVEL_INFO",
     "logManager":"LEVEL_WARNING",
     "clusterService":"LEVEL_ERROR",
     "syncService":"LEVEL_ERROR",
```

```
    "systemService":"LEVEL_INFO",
    "transactionManager":"LEVEL_ERROR",
    "transactionService":"LEVEL_ERROR",
    "transactionTimeout":"LEVEL_WARNING",
    "triggerService":"LEVEL_ERROR",
    "sessionTimeout":"LEVEL_WARNING",
    "replicationTimeout":"LEVEL_WARNING",
    "recoveryManager":"LEVEL_INFO",
    "eventEngine":"LEVEL_WARNING",
    "clusterOperation":"LEVEL_INFO",
    "ioMonitor":"LEVEL_WARNING"
  }
}
```

# B. JSON DATA MODEL

## B.1 Building And User Data

```
Group{
id: string
name: string
description: string
}



User{
  id: string
  email: string
  name: string
  groups: Groups[...]
}

Location{
  id: string
  x: double
  y: double
  z: double
}

InfrastructureType{
  id: string
  name: string
  description: string
}

Infrastructure{
  id: string
  name: string
  type: InfrastructureType{...}
  floor: integer
  geometry: Locations[...]
}
```

## B.2 Devices

```
PlatformType{
  id: string
  name: string
  description: string
}

Platform{
  id: string
  name: string
  type: PlatformType{..}
  owner: User{..}
  hashedMac: string
}
```

## B.3 Sensors and Observations

```
SensorType{
  id: string
  name: string
  description: string
  mobility: string
  payloadSchema: {...}
  captureFunctionality: string
}

SensorCoverage{
  id: string
  radius: float
  entitiesCovered: Infrastructure[..]
}

Sensor{
  id: string
  name: string
  description: string
  infrastructure: Infrastructure{...}
  type: SensorType{...}
  owner: User{..}
  coverage: Infrastructure[..]
  sensorConfig: {...}
}

Observation{
  id: string
  sensor: Sensor{..}
  timestamp: datetime
  payload: {...}
}
```

## B.4 Virtual Sensors and Semantic Observations

```
SemanticObservationType{
  id: string
  name: string
  description: string
  payloadSchema: {...}
}

VirtualSensorType{
  id: string
  name: string
  description: string
  inputType: SensorType{..}
  semanticObservationType: SemanticObservationType{...}
}

VirtualSensor{
  id: string
  name: string
  description: string
```

```
  type: VirtualSensorType{...}
  language: string
  projectName: string
}

SemanticObservation{
  id: string
  virtualSensor: VirtualSensor{...}
  timeStamp: datetime
  payload: {...}
  type: SemanticObservationType{..}
  semanticEntity: User{..} or Infrastructure{..}
}
```

# C. DATA AND QUERY GENERATION TOOL CONFIGURATION FILE

## C.1 Medium Dataset

```
[observation]
start_timestamp = 2017-11-08 00:00:00
days = 90
step = 300

[seed]
start_timestamp = 2017-11-08 00:00:00
days = 7
step = 600
speed-noise = 0.1
time-noise = 0.2
sensor-noise = 0.3
wemo = 10
temperature = 8
wifi = 64

[sensors]
wemo = 100
wifiap = 100
temperature = 100

[others]
pattern = random
users = 1360
rooms = 340
insert-test-data = 10
data-dir =
output-dir =

[query]
time-delta = 2
num-locations = 10
num-sensors = 5
runs = 5
output-dir =
```

## C.2 Large Dataset

```
[observation]
start_timestamp = 2017-11-08 00:00:00
days = 120
step = 200

[seed]
start_timestamp = 2017-11-08 00:00:00
days = 1
step = 600
speed-noise = 0.1
time-noise = 0.2
sensor-noise = 0.3
wemo = 10
temperature = 8
wifi = 64

[sensors]
wemo = 200
wifiap = 200
temperature = 200

[others]
pattern = random
users = 2400
rooms = 600
insert-test-data = 10
data-dir =
output-dir =

[query]
time-delta = 2
num-locations = 10
num-sensors = 5
runs = 3
output-dir =
```

## C.3 Multi Node Dataset

```
[observation]
start_timestamp = 2017-11-08 00:00:00
days = 120
step = 200

[seed]
start_timestamp = 2017-11-08 00:00:00
days = 7
step = 600
speed-noise = 0.1
time-noise = 0.2
sensor-noise = 0.3
wemo = 10
temperature = 8
wifi = 64

[sensors]
wemo = 1000
wifiap = 1000
temperature = 1000

[others]
pattern = random
users = 10000
rooms = 2500
```

```
insert-test-data = 10
data-dir =
output-dir =

[query]
time-delta = 2
num-locations = 10
num-sensors = 5
runs = 5
output-dir =
```

# D. QUERY MODELING

## D.1 MongoDB

```
Q1 :
MongoCollection<Document> collection = database.getCollection("Sensor");
MongoIterable<Document> iterable = collection.find(eq("id", sensorId))
       .projection(new Document("name", 1).append("_id", 0));

Q2:
MongoCollection<Document> collection = database.getCollection("Sensor");
MongoIterable<Document> iterable = collection.find(and(
       eq("type_.name", sensorTypeName),
       in("coverage.id", locationIds)))
       .projection(new Document("name", 1)
               .append("_id", 0)
               .append("id", 1));
Q3:
MongoCollection<Document> collection = database.getCollection("Observation");
MongoIterable<Document> iterable = collection.find(
       and(
       eq("sensor.id", sensorId),
       gt("timeStamp", startTime),
       lt("timeStamp", endTime)))
       .projection(new Document("timeStamp", 1)
               .append("_id", 0)
               .append("payload", 1)
.append("sensor.id", 1));

Q4:
MongoCollection<Document> collection = database.getCollection("Observation");
MongoIterable<Document> iterable = collection.find(
       and(
               in("sensor.id", sensorIds),
               gt("timeStamp", startTime),
               lt("timeStamp", endTime)))
       .projection(new Document("timeStamp", 1)
               .append("_id", 0)
               .append("sensor.id", 1)
               .append("payload, 1));

Q5:
MongoCollection<Document> collection = database.getCollection("Observation");
MongoIterable<Document> iterable = collection.find(
       and(
               eq("sensor.type_.name", sensorTypeName),
               gt("timeStamp", startTime),
               lt("timeStamp", endTime),
               gt(String.format("payload.%s", payloadAttribute), startPayloadValue),
               lt(String.format("payload.%s", payloadAttribute), endPayloadValue)))
       .projection(new Document("timeStamp", 1)
               .append("_id", 0)
               .append("sensor.id", 1)
               .append("payload", 1));


Q6:
MongoCollection<Document> collection = database.getCollection("Observation");

Bson match = match(and(
       in("sensor.id", sensorIds),
       gt("timeStamp", startTime),
       lt("timeStamp", endTime)));
```

```
Bson project = project(
       fields(
               excludeId(),
               include("sensor.id"),
               computed(
                       "date",
                       new Document("$dateToString", new Document("format", "%Y-%m-%d")
                               .append("date", "$timeStamp"))
               )
       )
);

Bson group1 = group(new Document("date", "$date").append("sensorId", "$sensor.id"),
       sum("count", 1));
Bson group2 = group(new Document("sensorId", "$_id.sensorId"), avg("averagePerDay", "$count"));

MongoIterable<Document> iterable = collection.aggregate(Arrays.asList(match, project, group1, group2));

Q7:
MongoCollection collection = database.getCollection("SemanticObservation");
Date startTime = date;
Calendar cal = Calendar.getInstance();
cal.setTime(date);
cal.add(Calendar.DATE, 1);
Date endTime = cal.getTime();

Bson match1 = match(and(
        eq("type_.name", "presence"),
        gt("timeStamp", startTime),
        lt("timeStamp", endTime),
        eq("payload.location", startLocation)
));

Bson lookUp = lookup("SemanticObservation", "semanticEntity.id",
        "semanticEntity.id", "semantics");

Bson unwind = unwind("$semantics");

Bson project1 = project(
        fields(
                excludeId(),
                include("timeStamp"),
                include("semantics"),
                include("semanticEntity"),
                include("payload"),
                computed("timeCheck",
                        new Document("$gt", Arrays.asList("$semantics.timeStamp", "$timeStamp")))
        )
);

Bson match2 = match(and(
        eq("semantics.type_.name", "presence"),
        lt("semantics.timeStamp", endTime),
        eq("semantics.payload.location", endLocation),
        eq("timeCheck", true)
));

Bson project2 = project(
        fields(
                include("semanticEntity.name")
        )
);

MongoIterable iterable = collection.aggregate(Arrays.asList(match1, lookUp, unwind, project1, match2, project2));
```

```
Q8:
MongoCollection collection = database.getCollection("SemanticObservation");
Date startTime = date;
Calendar cal = Calendar.getInstance();
cal.setTime(date);
cal.add(Calendar.DATE, 1);
Date endTime = cal.getTime();

Bson match1 = match(and(
        eq("type_.name", "presence"),
        gt("timeStamp", startTime),
        lt("timeStamp", endTime),
        eq("semanticEntity.id", userId)
));

Bson lookUp = lookup("SemanticObservation", "semanticEntity.id",
        "semanticEntity.id", "semantics");

Bson unwind = unwind("$semantics");

Bson project1 = project(
        fields(
                excludeId(),
                include("timeStamp"),
                include("semantics"),
                include("semanticEntity"),
                include("payload"),
                computed("timeCheck",
                        new Document("$eq", Arrays.asList("$semantics.timeStamp", "$timeStamp"))),
                computed("placeCheck",
                        new Document("$eq", Arrays.asList("$semantics.payload.location", "$payload.location")))
        )
);

Bson match2 = match(and(
        eq("semantics.type_.name", "presence"),
        lt("semantics.timeStamp", endTime),
        eq("timeCheck", true),
        eq("timeCheck", true)
));

Bson project2 = project(
        fields(
                include("semantics.semanticEntity.name"),
                include("payload.location")
        )
);

MongoIterable iterable = collection.aggregate(Arrays.asList(match1, lookUp, unwind, project1, match2, project2));

Q9:
MongoCollection<Document> collection = database.getCollection("SemanticObservation");

Bson lookUp1 = lookup("Infrastructure", "payload.location", "_id", "infra");

Bson unwind = unwind("$infra");

Bson match = match(and(
        eq("infra.type_.name", infraTypeName),
        eq("semanticEntity.id", userId)));

Bson project = project(
        fields(
```

```
                excludeId(),
                computed(
                        "date",
                        new Document("$dateToString", new Document("format", "%Y-%m-%d")
                                .append("date", "$timeStamp"))
                )
        )
);

Bson group1 = group(new Document("date", "$date"), sum("count", 10));
Bson group2 = group(null, avg("averageMinsPerDay", "$count"));

MongoIterable<Document> iterable = collection.aggregate(Arrays.asList(lookUp1, unwind, match, project,
        group1, group2));

Q10:
MongoCollection<Document> collection = database.getCollection("SemanticObservation");

Bson match = match(and(
        gt("timeStamp", startTime),
        lt("timeStamp", endTime),
        eq("type_.name", "occupancy")
));

Bson sort = sort(new Document("semanticEntity.id", 1).append("timeStamp", 1));

Bson project = project(
        fields(
                excludeId(),
                include("timeStamp"),
                include("semanticEntity.name"),
                include("payload.occupancy")
        )
);

MongoIterable<Document> iterable = collection.aggregate(Arrays.asList(match, sort, project));
```

## D.2 AsterixDB

```
Q1:
runTimedQuery(
   String.format("SELECT name FROM Sensor WHERE id = \"%s\";", sensorId), 1
);


Q2:
runTimedQuery(
       String.format("SELECT s.id, s.name FROM Sensor s WHERE s.type_.name=\"%s\" AND "
                       + "(SOME e IN s.coverage SATISFIES e.id IN {{"
                       + locationIds.stream().map(e -> "\"" + e + "\"" ).collect(Collectors.joining(","))
                       + "}});",
               sensorTypeName), 2
);

Q3:
runTimedQuery(
       String.format("SELECT timeStamp, sensor.id, payload FROM Observation WHERE sensor.id=\"%s\" "
               + "AND timeStamp >= datetime(\"%s\") AND timeStamp <= datetime(\"%s\");",
               sensorId, sdf.format(startTime), sdf.format(endTime)), 3
);

Q4:
```

```
runTimedQuery(
       String.format("SELECT timeStamp, sensor.id, payload FROM Observation WHERE sensor.id IN {{ "
                       + sensorIds.stream().map(e -> "\"" + e + "\"").collect(Collectors.joining(","))
                       + " }} AND timeStamp >= datetime(\"%s\") AND timeStamp <= datetime(\"%s\");",
               sdf.format(startTime), sdf.format(endTime)),  4
);

Q5:
runTimedQuery(
       String.format("SELECT timeStamp, sensor.id, payload " +
           "FROM Observation " +
           "WHERE sensor.type_.name = \"%s\" AND timeStamp >= datetime(\"%s\") AND " +
           "timeStamp <= datetime(\"%s\") " +
           "AND payload.%s >= %s AND payload.%s <= %s",
               sensorTypeName, sdf.format(startTime), sdf.format(endTime), payloadAttribute,
               startPayloadValue, payloadAttribute, endPayloadValue), 5
);


Q6:
runTimedQuery(
       String.format("SELECT obs.id , AVG(obs.count) FROM " +
               "(SELECT sensor.id , get_date_from_datetime(timeStamp), count(*)  AS count " +
               "FROM Observation " +
               "WHERE sensor.id IN {{ " +
               sensorIds.stream().map(e -> "\"" + e + "\"").collect(Collectors.joining(",")) +
               " }} AND timeStamp >= datetime(\"%s\") AND timeStamp <= datetime(\"%s\") " +
               "GROUP BY sensor.id, get_date_from_datetime(timeStamp)) AS obs GROUP BY obs.id",
               sdf.format(startTime), sdf.format(endTime)), 6
);

Q7:
runTimedQuery(
        String.format("SELECT s1.semanticEntity.name " +
                " FROM SemanticObservation s1, SemanticObservation s2 " +
                " WHERE get_date_from_datetime(s1.timeStamp) = date(\"%s\") AND " +
                " get_date_from_datetime(s2.timeStamp) = date(\"%s\") AND " +
                " s1.type_.name = s2.type_.name AND s2.type_.name = \"presence\" AND " +
                " s1.payload.location = \"%s\" AND s2.payload.location = \"%s\" " +
                " AND s1.timeStamp < s2.timeStamp AND s1.semanticEntity.id = s2.semanticEntity.id",
                dateOnlyFormat.format(date), dateOnlyFormat.format(date), startLocation, endLocation), 7
);

Q8:
runTimedQuery(
        String.format("SELECT s2.semanticEntity.name, s1.payload.location " +
                        " FROM SemanticObservation s1, SemanticObservation s2 " +
                        " WHERE get_date_from_datetime(s1.timeStamp) = date(\"%s\") AND " +
                        " get_date_from_datetime(s2.timeStamp) = date(\"%s\") AND " +
                        " s1.type_.name = s2.type_.name AND s2.type_.name = \"presence\" AND " +
                        " s1.payload.location = s2.payload.location " +
                        " AND s1.timeStamp = s2.timeStamp AND s1.semanticEntity.id = \"%s\" " +
                        " AND s2.semanticEntity.id != s1.semanticEntity.id",
                dateOnlyFormat.format(date), dateOnlyFormat.format(date), userId), 8
);

Q9:
runTimedQuery(
        String.format("SELECT AVG(groups.timeSpent) AS avgTimePerDay FROM " +
                        " (SELECT get_date_from_datetime(so.timeStamp), count(*)*10  AS timeSpent " +
                        " FROM SemanticObservation so, Infrastructure infra " +
                        " WHERE so.type_.name=\"presence\" AND so.semanticEntity.id=\"%s\" " +
                        " AND so.payload.location = infra.id AND infra.type_.name = \"%s\"" +
                        " GROUP BY get_date_from_datetime(so.timeStamp))  AS groups ",
```

```
                userId, infraTypeName), 9
);

Q10:
runTimedQuery(
        String.format("SELECT infra.name, (" +
                        "SELECT so.timeStamp, so.payload.occupancy " +
                        "FROM SemanticObservation so " +
                        "WHERE so.timeStamp > datetime(\"%s\") AND so.timeStamp < datetime(\"%s\") " +
                        "AND so.type_.name = \"occupancy\" AND so.semanticEntity.id = infra.id " +
                        "ORDER BY so.semanticEntity.id, so.timeStamp) AS histogram " +
                        "FROM Infrastructure infra", sdf.format(startTime), sdf.format(endTime)), 10
);
```

## D.3 PostgreSQL, CrateDB And SparkSQL

### *D.3.1 Mapping 1*

```
Q1:
query = "SELECT name FROM SENSOR WHERE id=?";

Q2:
query = "SELECT sen.name FROM SENSOR sen, SENSOR_TYPE st, " +
       "COVERAGE_INFRASTRUCTURE ci WHERE sen.SENSOR_TYPE_ID=st.id AND st.name=? " +
       "AND sen.id=ci.SENSOR_ID AND ci.INFRASTRUCTURE_ID=ANY(?)";

Q3:
query = "SELECT timeStamp, payload FROM OBSERVATION WHERE timestamp>? AND timestamp<? " +
       "AND SENSOR_ID=?";

Q4:
query = "SELECT timeStamp, payload FROM OBSERVATION WHERE timestamp>? AND timestamp<? " +
       "AND SENSOR_ID = ANY(?)";

Q5:
query = "SELECT timeStamp, payload FROM OBSERVATION o, SENSOR s, SENSOR_TYPE st  " +
       "WHERE s.id = o.sensor_id AND s.sensor_type_id=st.id AND st.name=? AND " +
       "timestamp>? AND timestamp<? ";
Along with the above query I am checking the payload value for each row returned if its in
the designated bound or not

Q6:
query = "SELECT obs.sensor_id, avg(counts) FROM " +
       "(SELECT sensor_id, date_trunc('day', timestamp), " +
       "count(*) as counts " +
       "FROM OBSERVATION WHERE timestamp>? AND timestamp<? " +
       "AND SENSOR_ID = ANY(?) GROUP BY sensor_id, date_trunc('day', timestamp)) " +
       "AS obs GROUP BY sensor_id";

Q7:
query = "SELECT u.name " +
       "FROM SEMANTIC_OBSERVATION s1, SEMANTIC_OBSERVATION s2, SEMANTIC_OBSERVATION_TYPE st, USERS u " +
       "WHERE date_trunc('day', s1.timeStamp) = ? " +
       "AND st.type = Presence AND st.id = s1.type_id AND st.id = s2.type_id " +
       "AND s1.semantic_entity_id = s2.semantic_entity_id \n" +
       "AND SUBSTRING (s1.payload, 0, 5) = ? AND SUBSTRING (s1.payload, 0, 5) = ? " +
       "AND s1.timeStamp < s2.timeStamp " +
       "AND s1.semantic_entity_id = u.id ";

Q8:
String query = "SELECT s2.semantic_entity_id " +
       "FROM SEMANTIC_OBSERVATION s1, SEMANTIC_OBSERVATION s2, SEMANTIC_OBSERVATION_TYPE st " +
```

```
       "WHERE date_trunc('day', s1.timeStamp) = ? " +
       "AND date_trunc('day', s2.timeStamp) = date_trunc('day', s1.timeStamp) " +
       "AND st.type = Presence AND s1.type_id = s2.type_id AND st.type_id = s1.type_id  " +
       "AND s1.semantic_entity_id = ? " +
       "AND SUBSTRING (s1.payload, 0, 5) = SUBSTRING (s2.payload, 0, 5) ";
Q9:
query = "SELECT Avg(timeSpent) as avgTimeSpent FROM " +
       " (SELECT date_trunc('day', so.timeStamp), count(*)*10 as timeSpent " +
       "  FROM SEMANTIC_OBSERVATION so, Infrastructure infra, SEMANTIC_OBSERVATION_TYPE st " +
       "  WHERE st.type = Presence AND so.type_id = st.type_id " +
       "  AND substring(so.payload, 0, 5) = infra.id " +
       "  AND infra.type = ? " +
       "  AND so.semantic_entity_id = ? " +
       "  GROUP BY  date(so.timeStamp))";

Q10:
query = "SELECT infra.name, so.timeStamp, so.payload.occupancy " +
       "FROM SEMANTIC_OBSERVATION so, INFRASTRUCTURE infra " +
       "WHERE so.timeStamp > ? AND so.timeStamp < ? " +
       "AND so.type_.name = Occupancy AND so.semanticEntity.id = infra.id " +
       "ORDER BY so.semantic_entity_id, so.timeStamp";
```

### *D.3.2 Mapping 2*

```
Q1:
query = "SELECT name FROM SENSOR WHERE id=?";

Q2:
query = "SELECT sen.name FROM SENSOR sen, SENSOR_TYPE st, " +
       "COVERAGE_INFRASTRUCTURE ci WHERE sen.SENSOR_TYPE_ID=st.id AND st.name=? " +
       "AND sen.id=ci.SENSOR_ID AND ci.INFRASTRUCTURE_ID=ANY(?)";

Q3:
if ("Thermometer".equals(typeId))
   query = "SELECT timeStamp, temperature FROM ThermometerObservation  WHERE timestamp>? AND timestamp<? " +
       "AND SENSOR_ID=?";
else if ("WeMo".equals(typeId))
   query = "SELECT timeStamp, currentMilliWatts, onTodaySeconds FROM WeMoObservation  WHERE timestamp>? " +
           "AND timestamp<? AND SENSOR_ID=?";
else if ("WiFiAP".equals(typeId))
   query = "SELECT timeStamp, clientId FROM WiFiAPObservation  WHERE timestamp>? AND timestamp<? " +
           "AND SENSOR_ID=?";


Q4:
if (!thermoSensors.isEmpty()) {
   query = "SELECT timeStamp, temperature FROM ThermometerObservation  WHERE timestamp>? AND timestamp<? " +
           "AND SENSOR_ID=ANY(?)";
   }
else if ("WeMo".equals(typeId)) {
   query = "SELECT timeStamp, currentMilliWatts, onTodaySeconds FROM WeMoObservation  WHERE timestamp>? " +
           "AND timestamp<? AND SENSOR_ID=ANY(?)";
}
else if ("WiFiAP".equals(typeId)) {
   query = "SELECT timeStamp, clientId FROM WiFiAPObservation  WHERE timestamp>? AND timestamp<? " +
           "AND SENSOR_ID=ANY(?)";
   }

Q5:
query = String.format("SELECT * FROM %sOBSERVATION o " +
       "WHERE timestamp>? AND timestamp<? AND %s>? AND %s<?", sensorTypeName,
       payloadAttribute, payloadAttribute);
```

```
Q6:
query = "SELECT ID, SENSOR_TYPE_ID FROM SENSOR WHERE ID=ANY(?)";
if (!thermoSensors.isEmpty()) {
   query = "SELECT obs.sensor_id, avg(counts) FROM " +
           "(SELECT sensor_id, date_trunc('day', timestamp), " +
           "count(*) as counts " +
           "FROM ThermometerObservation WHERE timestamp>? AND timestamp<? " +
           "AND SENSOR_ID = ANY(?) GROUP BY sensor_id, date_trunc('day', timestamp)) " +
           "AS obs GROUP BY sensor_id";
 }
else if ("WeMo".equals(typeId)) {
   query = "SELECT obs.sensor_id, avg(counts) FROM " +
           "(SELECT sensor_id, date_trunc('day', timestamp), " +
           "count(*) as counts " +
           "FROM WeMoObservation WHERE timestamp>? AND timestamp<? " +
           "AND SENSOR_ID = ANY(?) GROUP BY sensor_id, date_trunc('day', timestamp)) " +
           "AS obs GROUP BY sensor_id";
   }
else if ("WiFiAP".equals(typeId)) {
   query = "SELECT obs.sensor_id, avg(counts) FROM " +
           "(SELECT sensor_id, date_trunc('day', timestamp), " +
           "count(*) as counts " +
           "FROM WiFiAPObservation WHERE timestamp>? AND timestamp<? " +
           "AND SENSOR_ID = ANY(?) GROUP BY sensor_id, date_trunc('day', timestamp)) " +
           "AS obs GROUP BY sensor_id\begin{multicols*}{2}";
   }

Q7:
query = "SELECT u.name " +
      "FROM PRESENCE s1, PRESENCE s2, USERS u " +
      "WHERE date_trunc('day', s1.timeStamp) = ? " +
      "AND s1.semantic_entity_id = s2.semantic_entity_id " +
      "AND s1.location = ? AND s2.location = ? " +
      "AND s1.timeStamp < s2.timeStamp " +
      "AND s1.semantic_entity_id = u.id ";

Q8:
query = "SELECT s2.semantic_entity_id " +
       "FROM PRESENCE s1, PRESENCE s2 " +
       "WHERE date_trunc('day', s1.timeStamp) = ? " +
       "AND date_trunc('day', s2.timeStamp) = date_trunc('day', s1.timeStamp) " +
       "AND s1.semantic_entity_id = ? AND s1.semantic_entity_id != s2.semantic_entity_id " +
       "AND s1.location = s2.location ";

Q9:
query = "SELECT Avg(timeSpent) as avgTimeSpent FROM " +
       " (SELECT date_trunc('day', so.timeStamp), count(*)*10 as timeSpent " +
       "  FROM PRESENCE so, Infrastructure infra, Infrastructure_Type infraType " +
       "  WHERE so.location = infra.id " +
       "  AND infra.INFRASTRUCTURE_TYPE_ID = infraType.id AND infraType.name = ? " +
       "  AND so.semantic_entity_id = ? " +
       "  GROUP BY  date_trunc('day', so.timeStamp)) AS timeSpentPerDay";

Q10:
query = "SELECT infra.name, so.timeStamp, so.occupancy " +
       "FROM OCCUPANCY so, INFRASTRUCTURE infra " +
       "WHERE so.timeStamp > ? AND so.timeStamp < ? " +
       "AND so.semantic_entity_id = infra.id " +
       "ORDER BY so.semantic_entity_id, so.timeStamp";
```

## D.4 GridDB And InfluxDB

```
Q1:
```

```
Single Query on Sensor Container
"SELECT * FROM Sensor WHERE id='%s'"

Q2:
First Query on SensorType Container to fetch typeId for a given type name
"SELECT * FROM SensorType WHERE name='%s'"

Fetch From Sensor Container the sensors with given types.
"SELECT * FROM Sensor WHERE typeId='%s'"

For each Sensor check the coverage array and report the one that has the given roomIds.

Q3:
Get the particular sensor observation timeseries container name by concatenating given
sensorId to a predefined prefix.
Run a single query on the sensor observation timeseries container.
"SELECT * FROM %s WHERE timeStamp > TIMESTAMP('%s') " +
       "AND timeStamp < TIMESTAMP('%s')

Q4:
For each of the sensorIds in the list provided we get the particular sensor observation timeseries
container name by concatenating the sensorId to a predefined prefix.
Run a single query per sensorId on the corresponding observation timeseries container.
"SELECT * FROM %s WHERE timeStamp > TIMESTAMP('%s') " +
       "AND timeStamp < TIMESTAMP('%s')

Q5:
First Query on SensorType Container to fetch typeId for a given type name
"SELECT * FROM SensorType WHERE name='%s'"

Fetch From Sensor Container the sensors with given types.
"SELECT * FROM Sensor WHERE typeId='%s'"


Run a single query per sensorId on the corresponding observation timeseries container.
"SELECT * FROM %s WHERE timeStamp > TIMESTAMP('%s') " +
       "AND timeStamp < TIMESTAMP('%s') AND %s >= %s AND %s <= %s "

Q6:
Run a single query per sensorId on the corresponding observation timeseries container.
"SELECT * FROM %s WHERE timeStamp >= TIMESTAMP('%s') " +
       "AND timeStamp <= TIMESTAMP('%s')"

Run an in memory group by operation based on the the date portion of the timestamp fetched on each such list.

Q7:
- First query on Users table to fetch all users.
- For each user we get all the rows from the presence container of the user where the location is start location
  and the timestamp is between start and end time.
- Next for each such row found we find all the rows from the presence container of the user where location
  is end location and timestamp is between timestamp of outer row and end time.

Q8:
- First query on Users table to fetch user given by userId.
- From the presence container of that user fetch all tuples where timestamp is between start and end time.
- Next for each row fetched in previous step we query presence containers of all the users for tuples where
  timestamp is between start and end timestamp and location is same as the location of the original row.


Q9:
- First query on InfrastructureType table to fetch infrastructure type name given by infrastructureTypeId.
- Fetch all the infrastructures of type fetched in previous step from Infrastructure table.
- From the presence container of user given by userId, fetch all rows where timestamp is between
  start and end timestamp and the location is one the infrastructure fetched in previous step.
```

- For each set of rows fetched, do in memory hash based group by on the day of timestamp.

Q10:
- Fetch all infrastructures.
- From the occupancy container for each infrastructure fetch tuples where timestamp is between start and end day, in sorted order.

# E. QUERY PLANS

## E.1 PostgreSQL (Mapping 2)

```
Q1:
Index Scan using sensor_pkey on sensor  (cost=0.28..8.29 rows=1 width=12)
  Index Cond: ((id)::text = 'thermometer2'::text)

Q2:
Nested Loop  (cost=10.54..102.86 rows=2 width=12)
  ->  Hash Join  (cost=10.26..45.36 rows=60 width=47)
        Hash Cond: ((sen.sensor_type_id)::text = (st.id)::text)
        ->  Seq Scan on sensor sen  (cost=0.00..30.00 rows=1200 width=55)
        ->  Hash  (cost=10.25..10.25 rows=1 width=516)
              ->  Seq Scan on sensor_type st  (cost=0.00..10.25 rows=1 width=516)
                    Filter: ((name)::text = 'WiFiAP'::text)
  ->  Index Only Scan using coverage_infrastructure_pkey on coverage_infrastructure ci  (cost=0.28..0.95
  rows=1 width=33)
        Index Cond: ((infrastructure_id = ANY ('{5011,3059}'::text[])) AND (sensor_id = (sen.id)::text))
Q3:
Index Scan using temp_timestamp_idx on thermometerobservation  (cost=0.43..8.45 rows=1 width=12)
  Index Cond: (("timestamp" > '2017-11-01 00:00:00'::timestamp without time zone) AND ("timestamp" <
  '2017-11-07 10:10:10'::timestamp without time zone))
  Filter: ((sensor_id)::text = 'thermometer4'::text)

Q4:
Index Scan using temp_timestamp_idx on thermometerobservation  (cost=0.43..8.45 rows=1 width=12)
  Index Cond: (("timestamp" > '2017-11-07 11:12:59'::timestamp without time zone) AND ("timestamp" <
  '2017-11-07 15:15:59'::timestamp without time zone))
  Filter: ((sensor_id)::text = ANY ('{thermometer2,thermometer3,thermometer7}'::text[]))

Q5:
Index Scan using temp_timestamp_idx on thermometerobservation o  (cost=0.43..8.46 rows=1 width=85)
  Index Cond: (("timestamp" > '2017-11-01 00:00:00'::timestamp without time zone) AND ("timestamp" <
  '2017-11-07 11:10:10'::timestamp without time zone))
  Filter: ((temperature >= 0) AND (temperature <= 42))

Q6:
GroupAggregate  (cost=8.47..8.52 rows=1 width=68)
  Group Key: thermometerobservation.sensor_id
  ->  GroupAggregate  (cost=8.47..8.49 rows=1 width=52)
        Group Key: thermometerobservation.sensor_id, (date_trunc('day'::text,
        thermometerobservation."timestamp"))
        ->  Sort  (cost=8.47..8.47 rows=1 width=44)
              Sort Key: thermometerobservation.sensor_id, (date_trunc('day'::text,
              thermometerobservation."timestamp"))
              ->  Index Scan using temp_timestamp_idx on thermometerobservation  (cost=0.43..8.46 rows=1 width=44)
                    Index Cond: (("timestamp" > '2017-11-01 00:00:00'::timestamp without time zone) AND
                    ("timestamp" < '2017-11-07 05:10:10'::timestamp without time zone))
                    Filter: ((sensor_id)::text = ANY ('{thermometer1,thermometer5,thermometer7}'::text[]))

Q7:
Nested Loop  (cost=0.00..316078.69 rows=2 width=10)
  Join Filter: ((s1.semantic_entity_id)::text = (u.id)::text)
  ->  Nested Loop  (cost=0.00..316025.44 rows=2 width=72)
        Join Filter: ((s1."timestamp" < s2."timestamp") AND ((s1.semantic_entity_id)::text =
        (s2.semantic_entity_id)::text))
        ->  Seq Scan on presence s1  (cost=0.00..157972.17 rows=68 width=44)
              Filter: (((location)::text = '2059'::text) AND (date_trunc('day'::text, "timestamp") =
              '2017-11-08 00:00:00'::timestamp without time zone))
        ->  Materialize  (cost=0.00..157972.51 rows=68 width=44)
              ->  Seq Scan on presence s2  (cost=0.00..157972.17 rows=68 width=44)
                    Filter: (((location)::text = '2061'::text) AND (date_trunc('day'::text,
                    "timestamp") = '2017-11-08 00:00:00'::timestamp without time zone))
```

```
  ->  Materialize  (cost=0.00..28.50 rows=900 width=46)
        ->  Seq Scan on users u  (cost=0.00..24.00 rows=900 width=46)

Q8:
Nested Loop  (cost=0.43..158654.77 rows=12 width=15)
  Join Filter: ((s2.semantic_entity_id)::text = (u.id)::text)
  ->  Seq Scan on users u  (cost=0.00..24.00 rows=900 width=46)
  ->  Materialize  (cost=0.43..158468.80 rows=12 width=41)
        ->  Nested Loop  (cost=0.43..158468.74 rows=12 width=41)
              ->  Seq Scan on presence s1  (cost=0.00..157972.17 rows=25 width=49)
                    Filter: (((semantic_entity_id)::text = 'user1'::text) AND (date_trunc('day'::text,
                    "timestamp") = '2017-11-08 00:00:00'::timestamp without time zone))
              ->  Index Scan using presence_timestamp_idx on presence s2  (cost=0.43..19.85 rows=1 width=49)
                    Index Cond: ("timestamp" = s1."timestamp")
                    Filter: (((s1.semantic_entity_id)::text <> (semantic_entity_id)::text) AND
                    ((s1.location)::text = (location)::text))

Q9:
Aggregate  (cost=135228.38..135228.39 rows=1 width=32)
  ->  GroupAggregate  (cost=135224.84..135227.11 rows=101 width=16)
        Group Key: (date_trunc('day'::text, so."timestamp"))
        ->  Sort  (cost=135224.84..135225.09 rows=101 width=8)
              Sort Key: (date_trunc('day'::text, so."timestamp"))
              ->  Nested Loop  (cost=0.15..135221.48 rows=101 width=8)
                    Join Filter: ((infra.id)::text = (so.location)::text)
                    ->  Nested Loop  (cost=0.15..35.62 rows=7 width=5)
                          Join Filter: ((infra.infrastructure_type_id)::text = (infratype.id)::text)
                          ->  Index Scan using infrastructure_pkey on infrastructure infra
                          (cost=0.15..19.89 rows=340 width=12)
                          ->  Materialize  (cost=0.00..10.63 rows=1 width=516)
                                ->  Seq Scan on infrastructure_type infratype  (cost=0.00..10.62 rows=1 width=516)
                                      Filter: ((name)::text = 'Labs'::text)
                    ->  Materialize  (cost=0.00..134669.31 rows=5037 width=13)
                          ->  Seq Scan on presence so  (cost=0.00..134644.12 rows=5037 width=13)
                                Filter: ((semantic_entity_id)::text = 'user1'::text)

Q10:
Sort  (cost=31684.08..32158.32 rows=189694 width=22)
  Sort Key: so.semantic_entity_id, so."timestamp"
  ->  Hash Join  (cost=11.09..11162.26 rows=189694 width=22)
        Hash Cond: ((so.semantic_entity_id)::text = (infra.id)::text)
        ->  Index Scan using occupancy_timestamp_idx on occupancy so  (cost=0.44..8543.32 rows=189694 width=17)
              Index Cond: (("timestamp" > '2017-11-08 00:00:00'::timestamp without time zone) AND
              ("timestamp" < '2017-11-10 00:00:00'::timestamp without time zone))
        ->  Hash  (cost=6.40..6.40 rows=340 width=10)
              ->  Seq Scan on infrastructure infra  (cost=0.00..6.40 rows=340 width=10)
```

## E.2 SparkSQL (Mapping 2)

```
Q1:
*Project [name#3779]
+- *Filter (isnotnull(id#3778) && (id#3778 = thermometer2))
   +- HiveTableScan [name#3779, id#3778], HiveTableRelation `tippersdb`.`sensor`,
   org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [ID#3778, NAME#3779, INFRASTRUCTURE_ID#3780,
   USER_ID#3781, SENSOR_TYPE_ID#3782, SENSOR_CONFIG#3783]

Q2:
+- *SortMergeJoin [id#3855], [SENSOR_ID#3867], Inner
   :- *Sort [id#3855 ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(id#3855, 200)
   :     +- *Project [ID#3855, NAME#3856]
   :        +- *BroadcastHashJoin [SENSOR_TYPE_ID#3859], [id#3861], Inner, BuildRight
```

```
   :           :- *Filter (isnotnull(SENSOR_TYPE_ID#3859) && isnotnull(id#3855))
   :           :  +- HiveTableScan [ID#3855, NAME#3856, SENSOR_TYPE_ID#3859], HiveTableRelation
   `tippersdb`.`sensor`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [ID#3855, NAME#3856,
   INFRASTRUCTURE_ID#3857, USER_ID#3858, SENSOR_TYPE_ID#3859, SENSOR_CONFIG#3860]
   :           +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
   :              +- *Project [ID#3861]
   :                 +- *Filter ((isnotnull(name#3864) && (name#3864 = WiFiAP)) && isnotnull(id#3861))
   :                    +- HiveTableScan [ID#3861, name#3864], HiveTableRelation
   `tippersdb`.`sensor_type`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [ID#3861,
   DESCRIPTION#3862, MOBILITY#3863, NAME#3864, CAPTURE_FUNCTIONALITY#3865, PAYLOAD_SCHEMA#3866]
   +- *Sort [SENSOR_ID#3867 ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(SENSOR_ID#3867, 200)
         +- *Project [SENSOR_ID#3867]
            +- *Filter (((INFRASTRUCTURE_ID#3868 = 5011) || (INFRASTRUCTURE_ID#3868 = 3059)) &&
            isnotnull(SENSOR_ID#3867))
               +- HiveTableScan [SENSOR_ID#3867, INFRASTRUCTURE_ID#3868], HiveTableRelation
               `tippersdb`.`coverage_infrastructure`,
               org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [SENSOR_ID#3867,
               INFRASTRUCTURE_ID#3868]
Q5:
*Filter (((((isnotnull(temperature#4026) && isnotnull(timestamp#4027)) && (cast(timestamp#4027 as
string) > 2017-11-01 00:00:00.0)) && (cast(timestamp#4027 as string) < 2017-11-07 11:10:10.0)) &&
(temperature#4026 >= 0)) && (temperature#4026 <= 42))
+- HiveTableScan [id#4025, temperature#4026, timeStamp#4027, sensor_id#4028], HiveTableRelation
`tippersdb`.`thermometerobservation`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [id#4025,
temperature#4026, timeStamp#4027, sensor_id#4028]

Q7:
*Project [name#4089]
+- *SortMergeJoin [semantic_entity_id#4078], [id#4090], Inner
   :- *Project [semantic_entity_id#4078]
   :  +- *SortMergeJoin [semantic_entity_id#4078], [semantic_entity_id#4083], Inner, (timeStamp#4080 <
   timeStamp#4085)
   :     :- *Sort [semantic_entity_id#4078 ASC NULLS FIRST], false, 0
   :     :  +- Exchange hashpartitioning(semantic_entity_id#4078, 200)
   :     :     +- *Project [semantic_entity_id#4078, timeStamp#4080]
   :     :        +- *Filter (((((isnotnull(timeStamp#4080) && isnotnull(location#4079)) &&
   (cast(timeStamp#4080 as string) >= 2017-11-08 00:00:00.0)) && (cast(timeStamp#4080 as string) <=
   2017-11-09 00:00:00.0)) && (location#4079 = 2059)) && isnotnull(semantic_entity_id#4078))
   :     :           +- HiveTableScan [semantic_entity_id#4078, timeStamp#4080, location#4079],
   HiveTableRelation `tippersdb`.`presence`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe,
   [id#4077, semantic_entity_id#4078, location#4079, timeStamp#4080, virtual_sensor_id#4081]
   :     +- *Sort [semantic_entity_id#4083 ASC NULLS FIRST], false, 0
   :        +- Exchange hashpartitioning(semantic_entity_id#4083, 200)
   :           +- *Project [semantic_entity_id#4083, timeStamp#4085]
   :              +- *Filter ((((isnotnull(timeStamp#4085) && isnotnull(location#4084)) &&
   (cast(timeStamp#4085 as string) <= 2017-11-09 00:00:00.0)) && (location#4084 = 2061)) &&
   isnotnull(semantic_entity_id#4083))
   :                 +- HiveTableScan [semantic_entity_id#4083, timeStamp#4085, location#4084],
   HiveTableRelation `tippersdb`.`presence`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe,
   [id#4082, semantic_entity_id#4083, location#4084, timeStamp#4085, virtual_sensor_id#4086]
   +- *Sort [id#4090 ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(id#4090, 200)
         +- *Filter isnotnull(id#4090)
            +- HiveTableScan [NAME#4089, ID#4090], HiveTableRelation `tippersdb`.`users`,
            org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [EMAIL#4087, GOOGLE_AUTH_TOKEN#4088,
            NAME#4089, ID#4090]
Q8:
*Project [name#4106, location#4096]
+- *SortMergeJoin [semantic_entity_id#4100], [id#4107], Inner
   :- *Sort [semantic_entity_id#4100 ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(semantic_entity_id#4100, 200)
   :     +- *Project [location#4096, semantic_entity_id#4100]
   :        +- *SortMergeJoin [timeStamp#4097, location#4096], [timeStamp#4102, location#4101], Inner,
```

```
   NOT (semantic_entity_id#4095 = semantic_entity_id#4100)
   :           :- *Sort [timeStamp#4097 ASC NULLS FIRST, location#4096 ASC NULLS FIRST], false, 0
   :           :  +- Exchange hashpartitioning(timeStamp#4097, location#4096, 200)
   :           :     +- *Filter (((((isnotnull(semantic_entity_id#4095) && isnotnull(timeStamp#4097))
   && (cast(timeStamp#4097 as string) >= 2017-11-08 00:00:00.0)) && (cast(timeStamp#4097 as string) <=
   2017-11-09 00:00:00.0)) && (semantic_entity_id#4095 = user1)) && isnotnull(location#4096))
   :           :        +- HiveTableScan [semantic_entity_id#4095, location#4096, timeStamp#4097],
   HiveTableRelation `tippersdb`.`presence`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe,
   [id#4094, semantic_entity_id#4095, location#4096, timeStamp#4097, virtual_sensor_id#4098]
   :           +- *Sort [timeStamp#4102 ASC NULLS FIRST, location#4101 ASC NULLS FIRST], false, 0
   :              +- Exchange hashpartitioning(timeStamp#4102, location#4101, 200)
   :                 +- *Filter ((((isnotnull(semantic_entity_id#4100) && (cast(timeStamp#4102 as
   string) <= 2017-11-09 00:00:00.0)) && isnotnull(location#4101)) && isnotnull(timeStamp#4102)) &&
   (cast(timeStamp#4102 as string) >= 2017-11-08 00:00:00.0))
   :                    +- HiveTableScan [semantic_entity_id#4100, location#4101, timeStamp#4102],
   HiveTableRelation `tippersdb`.`presence`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe,
   [id#4099, semantic_entity_id#4100, location#4101, timeStamp#4102, virtual_sensor_id#4103]
   +- *Sort [id#4107 ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(id#4107, 200)
         +- *Filter isnotnull(id#4107)
            +- HiveTableScan [NAME#4106, ID#4107], HiveTableRelation `tippersdb`.`users`,
            org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [EMAIL#4104, GOOGLE_AUTH_TOKEN#4105,
            NAME#4106, ID#4107]

Q10:
*Project [name#4121, timeStamp#4119, occupancy#4118]
+- *Sort [semantic_entity_id#4117 ASC NULLS FIRST, timeStamp#4119 ASC NULLS FIRST], true, 0
   +- Exchange rangepartitioning(semantic_entity_id#4117 ASC NULLS FIRST, timeStamp#4119 ASC NULLS FIRST, 200)
      +- *Project [name#4121, timeStamp#4119, occupancy#4118, semantic_entity_id#4117]
         +- *SortMergeJoin [semantic_entity_id#4117], [id#4123], Inner
            :- *Sort [semantic_entity_id#4117 ASC NULLS FIRST], false, 0
            :  +- Exchange hashpartitioning(semantic_entity_id#4117, 200)
            :     +- *Filter (((isnotnull(timeStamp#4119) && (cast(timeStamp#4119 as string) >
            2017-11-08 00:00:00.0)) && (cast(timeStamp#4119 as string) < 2017-11-10 00:00:00.0)) &&
            isnotnull(semantic_entity_id#4117))
            :        +- HiveTableScan [semantic_entity_id#4117, occupancy#4118, timeStamp#4119],
            HiveTableRelation `tippersdb`.`occupancy`,
            org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, [id#4116, semantic_entity_id#4117,
            occupancy#4118, timeStamp#4119, virtual_sensor_id#4120]
            +- *Sort [id#4123 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(id#4123, 200)
                  +- *Filter isnotnull(id#4123)
                     +- HiveTableScan [NAME#4121, ID#4123], HiveTableRelation
                     `tippersdb`.`infrastructure`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe,
                     [NAME#4121, INFRASTRUCTURE_TYPE_ID#4122, ID#4123, FLOOR#4124]
```

## E.3 MongoDB (Mapping 2)

```
Q1:
{
  "queryPlanner": {
    "plannerVersion": 1,
    "namespace": "TippersDB.Sensor",
    "indexFilterSet": false,
    "parsedQuery": {
      "_id": {
        "$eq": "thermometer2"
      }
    },
    "winningPlan": {
      "stage": "PROJECTION",
      "transformBy": {
        "name": 1,
```

```
        "_id": 0
      },
      "inputStage": {
        "stage": "IDHACK"
      }
    },
    "rejectedPlans": []
  }
}

Q2:
{
  "queryPlanner": {
    "plannerVersion": 1,
    "namespace": "TippersDB.Sensor",
    "indexFilterSet": false,
    "parsedQuery": {
      "$and": [
        {
          "type_.name": {
            "$eq": "WiFiAP"
          }
        },
        {
          "coverage": {
            "$in": [
              "3059",
              "5011"
            ]
          }
        }
      ]
    },
    "winningPlan": {
      "stage": "PROJECTION",
      "transformBy": {
        "name": 1,
        "_id": 1
      },
      "inputStage": {
        "stage": "COLLSCAN",
        "filter": {
          "$and": [
            {
              "type_.name": {
                "$eq": "WiFiAP"
              }
            },
            {
              "coverage": {
                "$in": [
                  "3059",
                  "5011"
                ]
              }
            }
          ]
        },
        "direction": "forward"
      }
    },
    "rejectedPlans": []
  }
}
```

Q3:


```
{
  "queryPlanner": {
    "plannerVersion": 1,
    "namespace": "TippersDB.Observation",
    "indexFilterSet": false,
    "parsedQuery": {
      "$and": [
        {
          "sensorId": {
            "$eq": "thermometer4"
          }
        },
        {
          "timeStamp": {
            "$lt": {
              "$date": 1510078210000
            }
          }
        },
        {
          "timeStamp": {
            "$gt": {
              "$date": 1509519600000
            }
          }
        }
      ]
    },
    "winningPlan": {
      "stage": "PROJECTION",
      "transformBy": {
        "timeStamp": 1,
        "_id": 0,
        "payload": 1,
        "sensorId": 1
      },
      "inputStage": {
        "stage": "FETCH",
        "filter": {
          "sensorId": {
            "$eq": "thermometer4"
          }
        },
        "inputStage": {
          "stage": "IXSCAN",
          "keyPattern": {
            "timeStamp": 1
          },
          "indexName": "timeStamp_1",
          "isMultiKey": false,
          "multiKeyPaths": {
            "timeStamp": []
          },
          "isUnique": false,
          "isSparse": false,
          "isPartial": false,
          "indexVersion": 2,
          "direction": "forward",
          "indexBounds": {
            "timeStamp": [
              "(new Date(1509519600000), new Date(1510078210000))"
            ]
```

```
          }
        }
      }
    },
    "rejectedPlans": []
  }
}

Q4:
{
  "queryPlanner": {
    "plannerVersion": 1,
    "namespace": "TippersDB.Observation",
    "indexFilterSet": false,
    "parsedQuery": {
      "$and": [
        {
          "timeStamp": {
            "$lt": {
              "$date": 1510096559000
            }
          }
        },
        {
          "timeStamp": {
            "$gt": {
              "$date": 1510081979000
            }
          }
        },
        {
          "sensorId": {
            "$in": [
              "thermometer2",
              "thermometer3",
              "thermometer7"
            ]
          }
        }
      ]
    },
    "winningPlan": {
      "stage": "PROJECTION",
      "transformBy": {
        "timeStamp": 1,
        "_id": 0,
        "sensorId": 1,
        "payload": 1
      },
      "inputStage": {
        "stage": "FETCH",
        "filter": {
          "sensorId": {
            "$in": [
              "thermometer2",
              "thermometer3",
              "thermometer7"
            ]
          }
        },
        "inputStage": {
          "stage": "IXSCAN",
          "keyPattern": {
            "timeStamp": 1
```

```
          },
          "indexName": "timeStamp_1",
          "isMultiKey": false,
          "multiKeyPaths": {
            "timeStamp": []
          },
          "isUnique": false,
          "isSparse": false,
          "isPartial": false,
          "indexVersion": 2,
          "direction": "forward",
          "indexBounds": {
            "timeStamp": [
              "(new Date(1510081979000), new Date(1510096559000))"
            ]
          }
        }
      }
    },
    "rejectedPlans": []
  }
}

Q5:
{
  "queryPlanner": {
          "plannerVersion": 1,
          "namespace": "TippersDB.Observation",
          "indexFilterSet": false,
          "parsedQuery": {
            "$and": [
              {
                "payload.temperature": {
                  "$lte": 42
                }
              },
              {
                "timeStamp": {
                  "$lt": {
                    "$date": 1510081810000
                  }
                }
              },
              {
                "timeStamp": {
                  "$gt": {
                    "$date": 1509519600000
                  }
                }
              },
              {
                "payload.temperature": {
                  "$gte": 0
                }
              }
            ]
          },
          "winningPlan": {
            "stage": "FETCH",
            "filter": {
              "$and": [
                {
                  "payload.temperature": {
                    "$lte": 42
```

```
                }
              },
              {
                "payload.temperature": {
                  "$gte": 0
                }
              }
            ]
          },
          "inputStage": {
            "stage": "IXSCAN",
            "keyPattern": {
              "timeStamp": 1
            },
            "indexName": "timeStamp_1",
            "isMultiKey": false,
            "multiKeyPaths": {
              "timeStamp": []
            },
            "isUnique": false,
            "isSparse": false,
            "isPartial": false,
            "indexVersion": 2,
            "direction": "forward",
            "indexBounds": {
              "timeStamp": [
                "(new Date(1509519600000), new Date(1510081810000))"
              ]
            }
          }
        },
        "rejectedPlans": []
      }
    }
  }
}

Q6:
{
  "queryPlanner": {
          "plannerVersion": 1,
          "namespace": "TippersDB.Observation",
          "indexFilterSet": false,
          "parsedQuery": {
            "$and": [
              {
                "timeStamp": {
                  "$lt": {
                    "$date": 1510060210000
                  }
                }
              },
              {
                "timeStamp": {
                  "$gt": {
                    "$date": 1509519600000
                  }
                }
              },
              {
                "sensorId": {
                  "$in": [
                    "thermometer1",
                    "thermometer5",
```

```
                    "thermometer7"
                  ]
                }
              }
            ]
          },
          "winningPlan": {
            "stage": "FETCH",
            "filter": {
              "sensorId": {
                "$in": [
                  "thermometer1",
                  "thermometer5",
                  "thermometer7"
                ]
              }
            },
            "inputStage": {
              "stage": "IXSCAN",
              "keyPattern": {
                "timeStamp": 1
              },
              "indexName": "timeStamp_1",
              "isMultiKey": false,
              "multiKeyPaths": {
                "timeStamp": []
              },
              "isUnique": false,
              "isSparse": false,
              "isPartial": false,
              "indexVersion": 2,
              "direction": "forward",
              "indexBounds": {
                "timeStamp": [
                  "(new Date(1509519600000), new Date(1510060210000))"
                ]
              }
            }
          },
          "rejectedPlans": []
        }
      }
    }
}

Q7:
{
  "queryPlanner": {
          "plannerVersion": 1,
          "namespace": "TippersDB.SemanticObservation",
          "indexFilterSet": false,
          "parsedQuery": {
            "$and": [
              {
                "payload.location": {
                  "$eq": "2059"
                }
              },
              {
                "timeStamp": {
                  "$lt": {
                    "$date": 1510214400000
                  }
                }
```

```
            },
            {
              "timeStamp": {
                "$gt": {
                  "$date": 1510128000000
                }
              }
            }
          ]
        },
        "winningPlan": {
          "stage": "FETCH",
          "filter": {
            "payload.location": {
              "$eq": "2059"
            }
          },
          "inputStage": {
            "stage": "IXSCAN",
            "keyPattern": {
              "timeStamp": 1
            },
            "indexName": "timeStamp_1",
            "isMultiKey": false,
            "multiKeyPaths": {
              "timeStamp": []
            },
            "isUnique": false,
            "isSparse": false,
            "isPartial": false,
            "indexVersion": 2,
            "direction": "forward",
            "indexBounds": {
              "timeStamp": [
                "(new Date(1510128000000), new Date(1510214400000))"
              ]
            }
          }
        },
        "rejectedPlans": []
      }
    }
  }
}

Q8:
{
 "queryPlanner": {
          "plannerVersion": 1,
          "namespace": "TippersDB.SemanticObservation",
          "indexFilterSet": false,
          "parsedQuery": {
            "$and": [
              {
                "semanticEntityId": {
                  "$eq": "user1"
                }
              },
              {
                "timeStamp": {
                  "$lt": {
                    "$date": 1510214400000
                  }
                }
```

```
              },
              {
                "timeStamp": {
                  "$gt": {
                    "$date": 1510128000000
                  }
                }
              }
            ]
          },
          "winningPlan": {
            "stage": "FETCH",
            "filter": {
              "semanticEntityId": {
                "$eq": "user1"
              }
            },
            "inputStage": {
              "stage": "IXSCAN",
              "keyPattern": {
                "timeStamp": 1
              },
              "indexName": "timeStamp_1",
              "isMultiKey": false,
              "multiKeyPaths": {
                "timeStamp": []
              },
              "isUnique": false,
              "isSparse": false,
              "isPartial": false,
              "indexVersion": 2,
              "direction": "forward",
              "indexBounds": {
                "timeStamp": [
                  "(new Date(1510128000000), new Date(1510214400000))"
                ]
              }
            }
          },
          "rejectedPlans": []
        }
      }
    }
}

Q9:
{
  "queryPlanner": {
          "plannerVersion": 1,
          "namespace": "TippersDB.SemanticObservation",
          "indexFilterSet": false,
          "parsedQuery": {
            "semanticEntityId": {
              "$eq": "user1"
            }
          },
          "winningPlan": {
            "stage": "COLLSCAN",
            "filter": {
              "semanticEntityId": {
                "$eq": "user1"
              }
            },
            "direction": "forward"
```

```
          },
          "rejectedPlans": []
        }
      }
    }
}

Q10:
{
  "queryPlanner": {
          "plannerVersion": 1,
          "namespace": "TippersDB.SemanticObservation",
          "indexFilterSet": false,
          "parsedQuery": {
            "semanticEntityId": {
              "$eq": "user1"
            }
          },
          "winningPlan": {
            "stage": "COLLSCAN",
            "filter": {
              "semanticEntityId": {
                "$eq": "user1"
              }
            },
            "direction": "forward"
          },
          "rejectedPlans": []
        }
      }
    }
}
```

## E.4 CrateDB (Mapping 2)

## E.5 AsterixDB (Mapping 2)

```
Q1:
distribute result [$$7]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$7])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$7] <- [{"name": $$Sensor.getField(1)}]
      -- ASSIGN  |PARTITIONED|
        project ([$$Sensor])
        -- STREAM_PROJECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            unnest-map [$$8, $$Sensor] <- index-search("Sensor", 0, "TippersDB", "Sensor", FALSE, FALSE, 1, $$11, 1, $$
            -- BTREE_SEARCH  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                assign [$$11, $$12] <- ["thermometer2", "thermometer2"]
                -- ASSIGN  |PARTITIONED|
                  empty-tuple-source
                  -- EMPTY_TUPLE_SOURCE  |PARTITIONED|

Q2:
distribute result [$$17]
-- DISTRIBUTE_RESULT  |PARTITIONED|
```

```
exchange
-- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
  project ([$$17])
  -- STREAM_PROJECT  |PARTITIONED|
    assign [$$17] <- [{"id": $$27, "name": $$22}]
    -- ASSIGN  |PARTITIONED|
      project ([$$27, $$22])
      -- STREAM_PROJECT  |PARTITIONED|
        select ($$12)
        -- STREAM_SELECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            group by ([$$27 := $$18]) decor ([$$22]) {
                      aggregate [$$12] <- [non-empty-stream()]
                      -- AGGREGATE  |LOCAL|
                        select ($$11)
                        -- STREAM_SELECT  |LOCAL|
                          group by ([$$26 := $$24]) decor ([$$s; $$18; $$e; $$19; $$20; $$22; $$23]) {
                                    aggregate [$$11] <- [non-empty-stream()]
                                    -- AGGREGATE  |LOCAL|
                                      select (not(is-missing($$25)))
                                      -- STREAM_SELECT  |LOCAL|
                                        nested tuple source
                                        -- NESTED_TUPLE_SOURCE  |LOCAL|
                                 }
                          -- MICRO_PRE_CLUSTERED_GROUP_BY[$$24]  |LOCAL|
                            order (ASC, $$24)
                            -- IN_MEMORY_STABLE_SORT [$$24(ASC)]  |LOCAL|
                              nested tuple source
                              -- NESTED_TUPLE_SOURCE  |LOCAL|
                 }
            -- PRE_CLUSTERED_GROUP_BY[$$18]  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                order (ASC, $$18)
                -- STABLE_SORT [$$18(ASC)]  |PARTITIONED|
                  exchange
                  -- HASH_PARTITION_EXCHANGE [$$18]  |PARTITIONED|
                    project ([$$22, $$25, $$24, $$s, $$18, $$e, $$19, $$20, $$23])
                    -- STREAM_PROJECT  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        left outer join (eq($$e, $#1))
                        -- HYBRID_HASH_JOIN [$$e][$#1]  |PARTITIONED|
                          exchange
                          -- HASH_PARTITION_EXCHANGE [$$e]  |PARTITIONED|
                            assign [$$24] <- [create-query-uid()]
                            -- ASSIGN  |PARTITIONED|
                              unnest $$e <- scan-collection($$20)
                              -- UNNEST  |PARTITIONED|
                                select (eq($$19, "WiFiAP"))
                                -- STREAM_SELECT  |PARTITIONED|
                                  assign [$$19] <- [$$23.getField(1)]
                                  -- ASSIGN  |PARTITIONED|
                                    assign [$$23, $$22, $$20] <- [$$s.getField(2), $$s.getField(1), $$s.getField(5)]
                                    -- ASSIGN  |PARTITIONED|
                                      exchange
                                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                        data-scan []<-[$$18, $$s] <- TippersDB.Sensor
                                        -- DATASOURCE_SCAN  |PARTITIONED|
                                          exchange
                                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                            empty-tuple-source
                                            -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

```
                          exchange
                          -- HASH_PARTITION_EXCHANGE [$#1]  |PARTITIONED|
                            assign [$$25] <- [TRUE]
                            -- ASSIGN  |UNPARTITIONED|
                              unnest $#1 <- scan-collection(multiset: {{ "5011", "3059" }})
                              -- UNNEST  |UNPARTITIONED|
                                empty-tuple-source
                                -- EMPTY_TUPLE_SOURCE  |UNPARTITIONED|

Q3:
distribute result [$$14]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$14])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$14] <- [{"timeStamp": $$15, "sensorId": $$16, "payload": $$Observation.getField(2)}]
      -- ASSIGN  |PARTITIONED|
        select (and(ge($$15, 1509519600000000), le($$15, 1510078210000000), eq($$16, "thermometer4")))
        -- STREAM_SELECT  |PARTITIONED|
          assign [$$16, $$15] <- [$$Observation.getField(1), $$Observation.getField(3)]
          -- ASSIGN  |PARTITIONED|
            project ([$$Observation])
            -- STREAM_PROJECT  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                data-scan []<-[$$17, $$Observation] <- TippersDB.Observation
                -- DATASOURCE_SCAN  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    empty-tuple-source
                    -- EMPTY_TUPLE_SOURCE  |PARTITIONED|

Q4:
distribute result [$$17]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$17])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$17] <- [{"timeStamp": $$18, "sensorId": $$20, "payload": $$22}]
      -- ASSIGN  |PARTITIONED|
        project ([$$18, $$20, $$22])
        -- STREAM_PROJECT  |PARTITIONED|
          select ($$7)
          -- STREAM_SELECT  |PARTITIONED|
            project ([$$7, $$18, $$20, $$22])
            -- STREAM_PROJECT  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                group by ([$$24 := $$19]) decor ([$$18; $$20; $$22]) {
                          aggregate [$$7] <- [non-empty-stream()]
                          -- AGGREGATE  |LOCAL|
                            select (not(is-missing($$23)))
                            -- STREAM_SELECT  |LOCAL|
                              nested tuple source
                              -- NESTED_TUPLE_SOURCE  |LOCAL|
                       }
                -- PRE_CLUSTERED_GROUP_BY[$$19]  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    order (ASC, $$19)
                    -- STABLE_SORT [$$19(ASC)]  |PARTITIONED|
                      exchange
```

```
                    -- HASH_PARTITION_EXCHANGE [$$19]  |PARTITIONED|
                      project ([$$18, $$20, $$22, $$23, $$19])
                      -- STREAM_PROJECT  |PARTITIONED|
                        exchange
                        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                          left outer join (eq($$20, $#1))
                          -- HYBRID_HASH_JOIN [$$20][$#1]  |PARTITIONED|
                            exchange
                            -- HASH_PARTITION_EXCHANGE [$$20]  |PARTITIONED|
                              select (and(ge($$18, 1510081979000000), le($$18, 1510096559000000)))
                              -- STREAM_SELECT  |PARTITIONED|
                                project ([$$19, $$22, $$18, $$20])
                                -- STREAM_PROJECT  |PARTITIONED|
                                  assign [$$22, $$18, $$20] <- [$$Observation.getField(2), $$Observation.getField(3),
                                  -- ASSIGN  |PARTITIONED|
                                    exchange
                                    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                      data-scan []<-[$$19, $$Observation] <- TippersDB.Observation
                                      -- DATASOURCE_SCAN  |PARTITIONED|
                                        exchange
                                        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                          empty-tuple-source
                                          -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
                            exchange
                            -- HASH_PARTITION_EXCHANGE [$#1]  |PARTITIONED|
                              assign [$$23] <- [TRUE]
                              -- ASSIGN  |UNPARTITIONED|
                                unnest $#1 <- scan-collection(multiset: {{ "thermometer3", "thermometer2", "thermomet
                                -- UNNEST  |UNPARTITIONED|
                                  empty-tuple-source
                                  -- EMPTY_TUPLE_SOURCE  |UNPARTITIONED|

Q5:
distribute result [$$28]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$28])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$28] <- [{"timeStamp": $$30, "sensorId": $$31, "payload": $$35}]
      -- ASSIGN  |PARTITIONED|
        project ([$$30, $$31, $$35])
        -- STREAM_PROJECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            join (eq($$31, $$34))
            -- HYBRID_HASH_JOIN [$$31][$$34]  |PARTITIONED|
              exchange
              -- HASH_PARTITION_EXCHANGE [$$31]  |PARTITIONED|
                project ([$$30, $$31, $$35])
                -- STREAM_PROJECT  |PARTITIONED|
                  select (and(ge($$29, 0), le($$29, 42), le($$30, datetime: { 2017-11-07T11:10:10.000Z }), ge($$30, dat
                  -- STREAM_SELECT  |PARTITIONED|
                    project ([$$35, $$31, $$30, $$29])
                    -- STREAM_PROJECT  |PARTITIONED|
                      assign [$$31, $$30, $$29] <- [$$obs.getField(1), $$obs.getField(3), $$35.getField("temperature")]
                      -- ASSIGN  |PARTITIONED|
                        assign [$$35] <- [$$obs.getField(2)]
                        -- ASSIGN  |PARTITIONED|
                          project ([$$obs])
                          -- STREAM_PROJECT  |PARTITIONED|
                            exchange
                            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                              unnest-map [$$33, $$obs] <- index-search("Observation", 0, "TippersDB", "Observation", FA
```

```
                          -- BTREE_SEARCH  |PARTITIONED|
                            exchange
                            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                              order (ASC, $$42)
                              -- STABLE_SORT [$$42(ASC)]  |PARTITIONED|
                                exchange
                                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                  project ([$$42])
                                  -- STREAM_PROJECT  |PARTITIONED|
                                    exchange
                                    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                      unnest-map [$$41, $$42] <- index-search("obs_timestamp_idx", 0, "TippersDB",
                                      -- BTREE_SEARCH  |PARTITIONED|
                                        exchange
                                        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                          assign [$$39, $$40] <- [datetime: { 2017-11-01T00:00:00.000Z }, datetime:
                                          -- ASSIGN  |PARTITIONED|
                                            empty-tuple-source
                                            -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
            exchange
            -- HASH_PARTITION_EXCHANGE [$$34]  |PARTITIONED|
              project ([$$34])
              -- STREAM_PROJECT  |PARTITIONED|
                select (eq($$38.getField(1), "Thermometer"))
                -- STREAM_SELECT  |PARTITIONED|
                  project ([$$34, $$38])
                  -- STREAM_PROJECT  |PARTITIONED|
                    assign [$$38] <- [$$sen.getField(2)]
                    -- ASSIGN  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        data-scan []<-[$$34, $$sen] <- TippersDB.Sensor
                        -- DATASOURCE_SCAN  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            empty-tuple-source
                            -- EMPTY_TUPLE_SOURCE  |PARTITIONED|

Q6:
distribute result [$$43]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$43])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$43] <- [{"sensorId": $$sensorId, "$1": $$50}]
      -- ASSIGN  |PARTITIONED|
        exchange
        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
          group by ([$$sensorId := $$59]) decor ([]) {
                    aggregate [$$50] <- [agg-global-sql-avg($$58)]
                    -- AGGREGATE  |LOCAL|
                      nested tuple source
                      -- NESTED_TUPLE_SOURCE  |LOCAL|
                 }
          -- SORT_GROUP_BY[$$59]  |PARTITIONED|
            exchange
            -- HASH_PARTITION_EXCHANGE [$$59]  |PARTITIONED|
              group by ([$$59 := $$sensorId]) decor ([]) {
                        aggregate [$$58] <- [agg-local-sql-avg($$49)]
                        -- AGGREGATE  |LOCAL|
                          nested tuple source
                          -- NESTED_TUPLE_SOURCE  |LOCAL|
                       }
```

```
-- PRE_CLUSTERED_GROUP_BY[$$sensorId]  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$49, $$sensorId])
    -- STREAM_PROJECT  |PARTITIONED|
      exchange
      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
        group by ([$$sensorId := $$56; $#3 := $$57]) decor ([]) {
                  aggregate [$$49] <- [agg-sql-sum($$55)]
                  -- AGGREGATE  |LOCAL|
                    nested tuple source
                    -- NESTED_TUPLE_SOURCE  |LOCAL|
                }
       -- SORT_GROUP_BY[$$56, $$57]  |PARTITIONED|
          exchange
          -- HASH_PARTITION_EXCHANGE [$$56, $$57]  |PARTITIONED|
            group by ([$$56 := $$51; $$57 := $$45]) decor ([]) {
                      aggregate [$$55] <- [agg-sql-count(1)]
                      -- AGGREGATE  |LOCAL|
                        nested tuple source
                        -- NESTED_TUPLE_SOURCE  |LOCAL|
                     }
            -- SORT_GROUP_BY[$$51, $$45]  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                project ([$$51, $$45])
                -- STREAM_PROJECT  |PARTITIONED|
                  assign [$$45] <- [get-date-from-datetime($$47)]
                  -- ASSIGN  |PARTITIONED|
                    project ([$$51, $$47])
                    -- STREAM_PROJECT  |PARTITIONED|
                      select ($$17)
                      -- STREAM_SELECT  |PARTITIONED|
                        project ([$$17, $$51, $$47])
                        -- STREAM_PROJECT  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            group by ([$$54 := $$48]) decor ([$$51; $$47]) {
                                      aggregate [$$17] <- [non-empty-stream()]
                                      -- AGGREGATE  |LOCAL|
                                        select (not(is-missing($$53)))
                                        -- STREAM_SELECT  |LOCAL|
                                          nested tuple source
                                          -- NESTED_TUPLE_SOURCE  |LOCAL|
                                   }
                           -- PRE_CLUSTERED_GROUP_BY[$$48]  |PARTITIONED|
                             exchange
                             -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                               order (ASC, $$48)
                               -- STABLE_SORT [$$48(ASC)]  |PARTITIONED|
                                 exchange
                                 -- HASH_PARTITION_EXCHANGE [$$48]  |PARTITIONED|
                                   project ([$$51, $$47, $$53, $$48])
                                   -- STREAM_PROJECT  |PARTITIONED|
                                     exchange
                                     -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                       left outer join (eq($$51, $#8))
                                       -- HYBRID_HASH_JOIN [$$51][$#8]  |PARTITIONED|
                                         exchange
                                         -- HASH_PARTITION_EXCHANGE [$$51]  |PARTITIONED|
                                           select (and(ge($$47, datetime: { 2017-11-01T00:00:00.000Z }),
                                           -- STREAM_SELECT  |PARTITIONED|
                                             project ([$$48, $$47, $$51])
                                             -- STREAM_PROJECT  |PARTITIONED|
```

```
                                                    assign [$$47, $$51] <- [$$Observation.getField(3), $$Obse
                                                    -- ASSIGN  |PARTITIONED|
                                                      exchange
                                                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                                        unnest-map [$$48, $$Observation] <- index-search("Obs
                                                        -- BTREE_SEARCH  |PARTITIONED|
                                                          exchange
                                                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                                            order (ASC, $$63)
                                                            -- STABLE_SORT [$$63(ASC)]  |PARTITIONED|
                                                              exchange
                                                              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                                                project ([$$63])
                                                                -- STREAM_PROJECT  |PARTITIONED|
                                                                  exchange
                                                                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                                                    unnest-map [$$62, $$63] <- index-search("
                                                                    -- BTREE_SEARCH  |PARTITIONED|
                                                                      exchange
                                                                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                                                        assign [$$60, $$61] <- [datetime: { 2
                                                                        -- ASSIGN  |PARTITIONED|
                                                                          empty-tuple-source
                                                                          -- EMPTY_TUPLE_SOURCE  |PARTITIONED
                                                exchange
                                                -- HASH_PARTITION_EXCHANGE [$#8]  |PARTITIONED|
                                                  assign [$$53] <- [TRUE]
                                                  -- ASSIGN  |UNPARTITIONED|
                                                    unnest $#8 <- scan-collection(multiset: {{ "thermometer1",
                                                    -- UNNEST  |UNPARTITIONED|
                                                      empty-tuple-source
                                                      -- EMPTY_TUPLE_SOURCE  |UNPARTITIONED|
```

```
Q7:
distribute result [$$43]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$43])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$43] <- [{"name": $$60}]
      -- ASSIGN  |PARTITIONED|
        project ([$$60])
        -- STREAM_PROJECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            join (eq($$45, $$52))
            -- HYBRID_HASH_JOIN [$$45][$$52]  |PARTITIONED|
              exchange
              -- HASH_PARTITION_EXCHANGE [$$45]  |PARTITIONED|
                project ([$$60, $$45])
                -- STREAM_PROJECT  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    join (eq($$44, $$51))
                    -- HYBRID_HASH_JOIN [$$44][$$51]  |PARTITIONED|
                      exchange
                      -- HASH_PARTITION_EXCHANGE [$$44]  |PARTITIONED|
                        project ([$$45, $$44])
                        -- STREAM_PROJECT  |PARTITIONED|
                          select (lt($$47, $$46))
                          -- STREAM_SELECT  |PARTITIONED|
                            project ([$$47, $$45, $$44, $$46])
                            -- STREAM_PROJECT  |PARTITIONED|
```

```
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            join (and(eq($$55, $$45), eq($$56, $$44)))
            -- HYBRID_HASH_JOIN [$$55, $$56][$$45, $$44]  |PARTITIONED|
              exchange
              -- HASH_PARTITION_EXCHANGE [$$55, $$56]  |PARTITIONED|
                project ([$$55, $$47, $$56])
                -- STREAM_PROJECT  |PARTITIONED|
                  select (and(eq($$61.getField("location"), "2059"), eq(get-date-from-datetime($$47
                  -- STREAM_SELECT  |PARTITIONED|
                    project ([$$61, $$56, $$55, $$47])
                    -- STREAM_PROJECT  |PARTITIONED|
                      assign [$$61, $$56, $$55, $$47] <- [$$s1.getField(2), $$s1.getField(4), $$s1.
                      -- ASSIGN  |PARTITIONED|
                        project ([$$s1])
                        -- STREAM_PROJECT  |PARTITIONED|
                          assign [$$s1] <- [$$s2]
                          -- ASSIGN  |PARTITIONED|
                            exchange
                            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                              replicate
                              -- REPLICATE  |PARTITIONED|
                                exchange
                                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                  project ([$$s2])
                                  -- STREAM_PROJECT  |PARTITIONED|
                                    exchange
                                    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                      data-scan []<-[$$50, $$s2] <- TippersDB.SemanticObservation
                                      -- DATASOURCE_SCAN  |PARTITIONED|
                                        exchange
                                        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                          empty-tuple-source
                                          -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
              exchange
              -- HASH_PARTITION_EXCHANGE [$$45, $$44]  |PARTITIONED|
                project ([$$45, $$44, $$46])
                -- STREAM_PROJECT  |PARTITIONED|
                  select (and(eq($$62.getField("location"), "2061"), eq(get-date-from-datetime($$46
                  -- STREAM_SELECT  |PARTITIONED|
                    project ([$$62, $$45, $$46, $$44])
                    -- STREAM_PROJECT  |PARTITIONED|
                      assign [$$62, $$45, $$46, $$44] <- [$$s2.getField(2), $$s2.getField(5), $$s2.
                      -- ASSIGN  |PARTITIONED|
                        exchange
                        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                          replicate
                          -- REPLICATE  |PARTITIONED|
                            exchange
                            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                              project ([$$s2])
                              -- STREAM_PROJECT  |PARTITIONED|
                                exchange
                                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                  data-scan []<-[$$50, $$s2] <- TippersDB.SemanticObservation
                                  -- DATASOURCE_SCAN  |PARTITIONED|
                                    exchange
                                    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                      empty-tuple-source
                                      -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
        exchange
        -- HASH_PARTITION_EXCHANGE [$$51]  |PARTITIONED|
          project ([$$60, $$51])
          -- STREAM_PROJECT  |PARTITIONED|
```

```
                        assign [$$60] <- [$$se.getField(1)]
                        -- ASSIGN  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            data-scan []<-[$$51, $$se] <- TippersDB.User
                            -- DATASOURCE_SCAN  |PARTITIONED|
                              exchange
                              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                empty-tuple-source
                                -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
              exchange
              -- HASH_PARTITION_EXCHANGE [$$52]  |PARTITIONED|
                project ([$$52])
                -- STREAM_PROJECT  |PARTITIONED|
                  select (eq($$st.getField(1), "presence"))
                  -- STREAM_SELECT  |PARTITIONED|
                    exchange
                    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                      data-scan []<-[$$52, $$st] <- TippersDB.SemanticObservationType
                      -- DATASOURCE_SCAN  |PARTITIONED|
                        exchange
                        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                          empty-tuple-source
                          -- EMPTY_TUPLE_SOURCE  |PARTITIONED|

Q8:
distribute result [$$46]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$46])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$46] <- [{"name": $$63, "location": $$53}]
      -- ASSIGN  |PARTITIONED|
        project ([$$63, $$53])
        -- STREAM_PROJECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            join (eq($$49, $$57))
            -- HYBRID_HASH_JOIN [$$49][$$57]  |PARTITIONED|
              exchange
              -- HASH_PARTITION_EXCHANGE [$$49]  |PARTITIONED|
                project ([$$63, $$53, $$49])
                -- STREAM_PROJECT  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    join (eq($$47, $$56))
                    -- HYBRID_HASH_JOIN [$$47][$$56]  |PARTITIONED|
                      exchange
                      -- HASH_PARTITION_EXCHANGE [$$47]  |PARTITIONED|
                        project ([$$53, $$49, $$47])
                        -- STREAM_PROJECT  |PARTITIONED|
                          select (neq($$47, $$48))
                          -- STREAM_SELECT  |PARTITIONED|
                            project ([$$53, $$48, $$49, $$47])
                            -- STREAM_PROJECT  |PARTITIONED|
                              exchange
                              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                join (and(eq($$60, $$49), eq($$53, $$61), eq($$51, $$50)))
                                -- HYBRID_HASH_JOIN [$$60, $$53, $$51][$$49, $$61, $$50]  |PARTITIONED|
                                  exchange
                                  -- HASH_PARTITION_EXCHANGE [$$60, $$53, $$51]  |PARTITIONED|
                                    project ([$$53, $$60, $$51, $$48])
                                    -- STREAM_PROJECT  |PARTITIONED|
```

```
                    assign [$$53] <- [$$s1.getField(2).getField("location")]
                    -- ASSIGN  |PARTITIONED|
                      select (and(eq($$48, "user1"), eq(get-date-from-datetime($$51), "date": { 2017-
                      -- STREAM_SELECT  |PARTITIONED|
                        project ([$$s1, $$60, $$51, $$48])
                        -- STREAM_PROJECT  |PARTITIONED|
                          assign [$$s1, $$60, $$51, $$48] <- [$$s2, $$49, $$50, $$47]
                          -- ASSIGN  |PARTITIONED|
                            exchange
                            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                              replicate
                              -- REPLICATE  |PARTITIONED|
                                exchange
                                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                  assign [$$49, $$50, $$47] <- [$$s2.getField(5), $$s2.getField(3), $
                                  -- ASSIGN  |PARTITIONED|
                                    project ([$$s2])
                                    -- STREAM_PROJECT  |PARTITIONED|
                                      exchange
                                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                        data-scan []<-[$$55, $$s2] <- TippersDB.SemanticObservation
                                        -- DATASOURCE_SCAN  |PARTITIONED|
                                          exchange
                                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                            empty-tuple-source
                                            -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
                  exchange
                  -- HASH_PARTITION_EXCHANGE [$$49, $$61, $$50]  |PARTITIONED|
                    project ([$$49, $$47, $$61, $$50])
                    -- STREAM_PROJECT  |PARTITIONED|
                      assign [$$61] <- [$$s2.getField(2).getField("location")]
                      -- ASSIGN  |PARTITIONED|
                        select (eq(get-date-from-datetime($$50), "date": { 2017-11-08 }))
                        -- STREAM_SELECT  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            replicate
                            -- REPLICATE  |PARTITIONED|
                              exchange
                              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                assign [$$49, $$50, $$47] <- [$$s2.getField(5), $$s2.getField(3), $$s2.
                                -- ASSIGN  |PARTITIONED|
                                  project ([$$s2])
                                  -- STREAM_PROJECT  |PARTITIONED|
                                    exchange
                                    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                      data-scan []<-[$$55, $$s2] <- TippersDB.SemanticObservation
                                      -- DATASOURCE_SCAN  |PARTITIONED|
                                        exchange
                                        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                          empty-tuple-source
                                          -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
exchange
-- HASH_PARTITION_EXCHANGE [$$56]  |PARTITIONED|
  project ([$$63, $$56])
  -- STREAM_PROJECT  |PARTITIONED|
    assign [$$63] <- [$$se.getField(1)]
    -- ASSIGN  |PARTITIONED|
      exchange
      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
        data-scan []<-[$$56, $$se] <- TippersDB.User
        -- DATASOURCE_SCAN  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
```

```
                              empty-tuple-source
                              -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
            exchange
            -- HASH_PARTITION_EXCHANGE [$$57]  |PARTITIONED|
              project ([$$57])
              -- STREAM_PROJECT  |PARTITIONED|
                select (eq($$st.getField(1), "presence"))
                -- STREAM_SELECT  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    data-scan []<-[$$57, $$st] <- TippersDB.SemanticObservationType
                    -- DATASOURCE_SCAN  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        empty-tuple-source
                        -- EMPTY_TUPLE_SOURCE  |PARTITIONED|

Q9:
distribute result [$$47]
-- DISTRIBUTE_RESULT  |UNPARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |UNPARTITIONED|
    project ([$$47])
    -- STREAM_PROJECT  |UNPARTITIONED|
      assign [$$47] <- [{"avgTimePerDay": $$55}]
      -- ASSIGN  |UNPARTITIONED|
        aggregate [$$55] <- [agg-global-sql-avg($$67)]
        -- AGGREGATE  |UNPARTITIONED|
          exchange
          -- RANDOM_MERGE_EXCHANGE  |PARTITIONED|
            aggregate [$$67] <- [agg-local-sql-avg($$45)]
            -- AGGREGATE  |PARTITIONED|
              project ([$$45])
              -- STREAM_PROJECT  |PARTITIONED|
                assign [$$45] <- [numeric-multiply($$54, 10)]
                -- ASSIGN  |PARTITIONED|
                  project ([$$54])
                  -- STREAM_PROJECT  |PARTITIONED|
                    exchange
                    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                      group by ([$#2 := $$66]) decor ([]) {
                                aggregate [$$54] <- [agg-sql-sum($$65)]
                                -- AGGREGATE  |LOCAL|
                                  nested tuple source
                                  -- NESTED_TUPLE_SOURCE  |LOCAL|
                             }
                     -- SORT_GROUP_BY[$$66]  |PARTITIONED|
                       exchange
                       -- HASH_PARTITION_EXCHANGE [$$66]  |PARTITIONED|
                         group by ([$$66 := $$48]) decor ([]) {
                                   aggregate [$$65] <- [agg-sql-count(1)]
                                   -- AGGREGATE  |LOCAL|
                                     nested tuple source
                                     -- NESTED_TUPLE_SOURCE  |LOCAL|
                                }
                         -- SORT_GROUP_BY[$$48]  |PARTITIONED|
                           exchange
                           -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                             project ([$$48])
                             -- STREAM_PROJECT  |PARTITIONED|
                               assign [$$48] <- [get-date-from-datetime($$61)]
                               -- ASSIGN  |PARTITIONED|
                                 project ([$$61])
                                 -- STREAM_PROJECT  |PARTITIONED|
```

```
exchange
-- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
  join (eq($$52, $$51))
  -- HYBRID_HASH_JOIN [$$52][$$51]  |PARTITIONED|
    exchange
    -- HASH_PARTITION_EXCHANGE [$$52]  |PARTITIONED|
      project ([$$61, $$52])
      -- STREAM_PROJECT  |PARTITIONED|
        exchange
        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
          join (eq($$56, $$50))
          -- HYBRID_HASH_JOIN [$$56][$$50]  |PARTITIONED|
            exchange
            -- HASH_PARTITION_EXCHANGE [$$56]  |PARTITIONED|
              project ([$$61, $$52, $$56])
              -- STREAM_PROJECT  |PARTITIONED|
                assign [$$56] <- [$$so.getField(2).getField("location")]
                -- ASSIGN  |PARTITIONED|
                  select (eq($$so.getField(4), "user1"))
                  -- STREAM_SELECT  |PARTITIONED|
                    assign [$$61, $$52] <- [$$so.getField(3), $$so.getField(5)]
                    -- ASSIGN  |PARTITIONED|
                      project ([$$so])
                      -- STREAM_PROJECT  |PARTITIONED|
                        exchange
                        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                          data-scan []<-[$$49, $$so] <- TippersDB.SemanticObservati
                          -- DATASOURCE_SCAN  |PARTITIONED|
                            exchange
                            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                              empty-tuple-source
                              -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
            exchange
            -- HASH_PARTITION_EXCHANGE [$$50]  |PARTITIONED|
              project ([$$50])
              -- STREAM_PROJECT  |PARTITIONED|
                select (eq($$63.getField(1), "Labs"))
                -- STREAM_SELECT  |PARTITIONED|
                  project ([$$50, $$63])
                  -- STREAM_PROJECT  |PARTITIONED|
                    assign [$$63] <- [$$infra.getField(2)]
                    -- ASSIGN  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        data-scan []<-[$$50, $$infra] <- TippersDB.Infrastructure
                        -- DATASOURCE_SCAN  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            empty-tuple-source
                            -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
    exchange
    -- HASH_PARTITION_EXCHANGE [$$51]  |PARTITIONED|
      project ([$$51])
      -- STREAM_PROJECT  |PARTITIONED|
        select (eq($$st.getField(1), "presence"))
        -- STREAM_SELECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            data-scan []<-[$$51, $$st] <- TippersDB.SemanticObservationType
            -- DATASOURCE_SCAN  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                empty-tuple-source
                -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

```
Q10:
distribute result [$$32]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$32])
    -- STREAM_PROJECT  |PARTITIONED|
      assign [$$32] <- [{"name": $$45, "histogram": $$31}]
      -- ASSIGN  |PARTITIONED|
        project ([$$31, $$45])
        -- STREAM_PROJECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            group by ([$$48 := $$34]) decor ([$$45]) {
                      aggregate [$$31] <- [listify({"timeStamp": $$33, "occupancy": $$44})]
                      -- AGGREGATE  |LOCAL|
                        select (not(is-missing($$47)))
                        -- STREAM_SELECT  |LOCAL|
                          nested tuple source
                          -- NESTED_TUPLE_SOURCE  |LOCAL|
                   }
            -- PRE_CLUSTERED_GROUP_BY[$$34]  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                order (ASC, $$34) (ASC, $$37) (ASC, $$33)
                -- STABLE_SORT [$$34(ASC), $$37(ASC), $$33(ASC)]  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    left outer join (eq($$37, $$34))
                    -- HYBRID_HASH_JOIN [$$34][$$37]  |PARTITIONED|
                      exchange
                      -- HASH_PARTITION_EXCHANGE [$$34]  |PARTITIONED|
                        project ([$$45, $$34])
                        -- STREAM_PROJECT  |PARTITIONED|
                          assign [$$45] <- [$$infra.getField(1)]
                          -- ASSIGN  |PARTITIONED|
                            exchange
                            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                              data-scan []<-[$$34, $$infra] <- TippersDB.Infrastructure
                              -- DATASOURCE_SCAN  |PARTITIONED|
                                exchange
                                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                  empty-tuple-source
                                  -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
                      exchange
                      -- HASH_PARTITION_EXCHANGE [$$37]  |PARTITIONED|
                        assign [$$47] <- [TRUE]
                        -- ASSIGN  |PARTITIONED|
                          project ([$$33, $$44, $$37])
                          -- STREAM_PROJECT  |PARTITIONED|
                            exchange
                            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                              join (eq($$39, $$36))
                              -- HYBRID_HASH_JOIN [$$39][$$36]  |PARTITIONED|
                                exchange
                                -- HASH_PARTITION_EXCHANGE [$$39]  |PARTITIONED|
                                  project ([$$33, $$44, $$37, $$39])
                                  -- STREAM_PROJECT  |PARTITIONED|
                                    assign [$$44] <- [$$so.getField(2).getField("occupancy")]
                                    -- ASSIGN  |PARTITIONED|
                                      select (and(gt($$33, datetime: { 2017-11-08T00:00:00.000Z }), lt($$33, datetime:
                                      -- STREAM_SELECT  |PARTITIONED|
                                        assign [$$37, $$39, $$33] <- [$$so.getField(4), $$so.getField(5), $$so.getFiel
```

```
          -- ASSIGN  |PARTITIONED|
            project ([$$so])
            -- STREAM_PROJECT  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                unnest-map [$$35, $$so] <- index-search("SemanticObservation", 0, "Tipper
                -- BTREE_SEARCH  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    order (ASC, $$52)
                    -- STABLE_SORT [$$52(ASC)]  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        project ([$$52])
                        -- STREAM_PROJECT  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            unnest-map [$$51, $$52] <- index-search("so_timestamp_idx", 0
                            -- BTREE_SEARCH  |PARTITIONED|
                              exchange
                              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                                assign [$$49, $$50] <- [datetime: { 2017-11-08T00:00:00.0
                                -- ASSIGN  |PARTITIONED|
                                  empty-tuple-source
                                  -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
exchange
-- HASH_PARTITION_EXCHANGE [$$36]  |PARTITIONED|
  project ([$$36])
  -- STREAM_PROJECT  |PARTITIONED|
    select (eq($$st.getField(1), "occupancy"))
    -- STREAM_SELECT  |PARTITIONED|
      exchange
      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
        data-scan []<-[$$36, $$st] <- TippersDB.SemanticObservationType
        -- DATASOURCE_SCAN  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            empty-tuple-source
            -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```